2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering

FloodShield: Securing the SDN Infrastructure Against Denial-of-Service Attacks

Menghao Zhang, Jun Bi*, Jiasong Bai, Guanyu Li
Institute for Network Sciences and Cyberspace, Tsinghua University
Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)
*Corresponding author: junbi@tsinghua.edu.cn

***Abstract*—Software-Defined Networking (SDN) has attracted great attention from both academia and industry. However, the deployment of SDN has faced some critical security issues, such as *Denial-of-Service* (DoS) attacks on the SDN infrastructure. One such DoS attack is the *data-to-control plane saturation attack*, where an attacker floods a large number of packets to trigger massive table-misses and `packet-in` messages in the data plane. This attack can exhaust resources of different components of the SDN infrastructure, including TCAM and buffer memory in the data plane, bandwidth of the control channel, and CPU cycles of the controller. In this paper, we analyze the vulnerability of SDN against the data-to-control plane saturation attack extensively and design FloodShield, a comprehensive, deployable and lightweight SDN defense framework to mitigate this dedicated DoS attack. FloodShield combines the following two techniques: 1) *source address validation* which filters forged packets directly in the data plane, and 2) *stateful packet supervision* which monitors traffic states of real addresses and performs dynamic countermeasures based on evaluation scores and network resource usages. Implementations and experiments demonstrate that, in comparison with state-of-the-art defense frameworks, FloodShield provides effective protection for all three components of the SDN infrastructure – data plane, control channel and control plane – with less resource consumption.**

***Index Terms*—Software-Defined Networking, Denial-of-Service Attack, Source Address Validation, Stateful Packet Supervision**

I. Introduction

Software-Defined Networking (SDN) is an emerging network architecture which simplifies network management and optimization with fine-grained and centralized control. By decoupling the control plane and the data plane, it has enabled unprecedented programmability, automation, and innovations in computer networks. Thus, the typical SDN infrastructure consists of three major components: the control plane, the data plane and a control channel where the two planes can communicate through standard protocols.

As the *de facto* standard SDN protocol[1], OpenFlow [1] introduces a *reactive* packet processing mechanism with the *match-action* paradigm: an OpenFlow switch processes packets based on flow tables and when no flow entries in the local flow table match a certain packet (known as a *table-miss*), the switch encapsulates this packet in a `packet-in` message and forwards it to the controller for further processing. After the controller receives the request message, it computes new flow rules and installs them on switches with `flow-mod` messages.

This reactive packet processing mechanism enables SDN to quickly adapt to network dynamics, however, it may also be a new vulnerability of the SDN infrastructure. When numerous packets arrive at a switch where no matching flow entries can be found, a large number of `packet-in` messages would be sent to the controller through the control channel. These massive `packet-in` messages can overwhelm the controller, overflow buffer memory and flow tables on switches, and congest the control channel, resulting in performance degradation and even collapse of the entire network.

This vulnerability can be exploited to conduct the *data-to-control-plane saturation attack* [2], [3], [4], a dedicated *Denial-of-Service (DoS) Attack* against the SDN infrastructure. By controlling several zombie hosts, an attacker generates a large number of malicious packets whose packet headers are filled with deliberately forged values. These malicious packets will trigger a great number of table-misses and `packet-in` messages, which can paralyze the SDN infrastructure by exhausting available resources in the data plane, the control plane or the control channel.

Several previous studies have been devoted to defending against this attack. AVANT-GUARD [2] modifies the switch design to mitigate TCP-based flooding attack. FloodGuard [3] introduces a *Data Plane Cache* to protect the controller. FloodDefender [4] adopts three techniques and designs four complicated modules in the controller to mitigate the SDN-aimed DoS attacks. Unfortunately, to the best of our knowledge, there has not been a solution which satisfies the following three requirements/principles simultaneously, which we feel are important to a practical DoS defense framework.

1) **Provide *overall* protection for the SDN infrastructure**: Existing works such as AVANT-GUARD and FloodGuard mainly focus on protecting SDN controllers while ignoring the data plane and the control channel. However, even if the controller is well protected, the attack can still take effect when there are no proper protections for the data plane and the controller channel [5]. Besides, since Ternary Content Addressable Memory (TCAM) in a switch is expensive and power-hungry [6], [7], preventing switches from the saturation attack

[1]Since OpenFlow is one of the most representative forms of SDN, in this paper, we use SDN and OpenFlow interchangeably for brevity.

2324-9013/18/31.00 ©2018 IEEE
DOI 10.1109/TrustCom/BigDataSE.2018.00101

and overflow attack [5] is crucial to guarantee service quality and to save energy.

2) **Easy for real-world deployment**: As SDN has been widely deployed in many enterprises/campuses/ISPs [8], [9], addressing this attack without re-designing the existing SDN infrastructure or introducing additional devices is both essential and cost-effective. AVANT-GUARD and FloodGuard fail to satisfy this requirement, and both approaches need to use additional devices to mitigate this attack.
3) **Lightweight and incur reasonable overheads to the controller**: As the core component of the SDN infrastructure, the controller plays a crucial role in the SDN architecture and has undertaken the greatest pressure. Existing countermeasures, such as FloodDefender, put all the burden on the controller and deal with table-misses and `packet-in` messages with modules on the controller platform, which would inevitably compromise the effects of protection. A lightweight solution with reasonable overheads to the controller is in urgent need.

To fulfill these requirements, this paper proposes *FloodShield*, a novel and effective SDN defense framework against the data-to-control plane saturation attacks. By combining two novel techniques, *source address validation* and *stateful packet supervision*, FloodShield provides a comprehensive protection for all three components of the SDN infrastructure. Since our framework does not modify OpenFlow switches or introduce additional devices, it can be easily deployed in the existing OpenFlow networks. Besides, with an effective coordination mechanism between the control plane and the data plane, it is lightweight and incurs reasonable burden to the controller.

To summarize, the contributions of this paper are as follows:

- We extensively analyze the vulnerability of the SDN infrastructure against the data-to-control plane saturation attack and the drawbacks of the state-of-the-art approaches. (§II)
- We propose and design FloodShield, a comprehensive, deployable and lightweight defense framework for the SDN infrastructure with source address validation and stateful packet supervision. (§IV)
- We implement a prototype of FloodShield and evaluate it in different scenarios. Evaluations demonstrate that, in comparison with the state-of-the-art frameworks, FloodShield provides effective protection for the SDN infrastructure with less resource consumption. (§V)

The rest of this paper is structured as follows. We state the problem in Section II, the observation and motivation in Section III. Section IV describes the detailed design of FloodShield and Section V illustrates the implementation and evaluation. We make some discussions in Section VI and introduce the related work in Section VII. Finally, in Section VIII, we conclude our work and make some prospects.

II. Problem Statement

In this section, we first describe the adversary model of the data-to-control plane saturation attack, and then present the drawbacks of the existing state-of-the-art defense approaches.

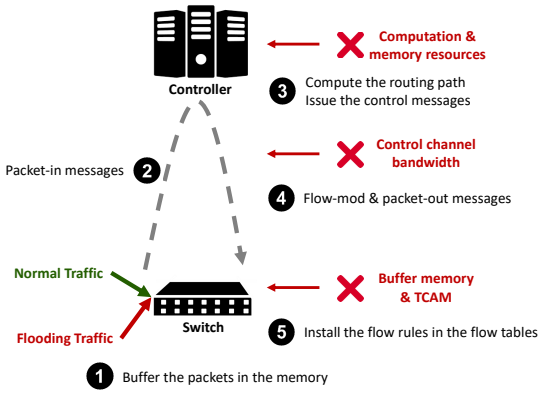


Fig. 1. Attack Process and Threatened Resources

A. Adversary Model

An attacker commits the data-to-control plane saturation attack by producing a large number of short-flows by controlling a number of zombie hosts in an SDN-enabled network. The attack traffic is mixed with normal traffic, making it difficult to be identified. With the reactive packet processing and fine-grained flow control mechanism taken by the existing mainstream SDN controllers, as we evidenced in Table I, the unmatched packets in the data plane would be delivered to the controller directly and processed by the corresponding logics. As a result, the data plane, the control channel and the control plane would quickly suffer from the attack and soon the SDN system could not provide any service for benign traffic.

We start from a simplified motivating scenario to illustrate how an adversary attacks the SDN infrastructure. As shown in Figure 1, when a new packet arrives at a switch where there is no matching flow entry in the local flow tables, the switch will store the packet in its buffer memory and send a `packet-in` message to the controller. If the buffer memory is not full, the message only contains the packet header but if the buffer memory is full, the message will contain the whole packet. After the controller receives the message, it computes the route and takes the corresponding actions on the switches through control messages like `flow-mod` and `packet-out`. Then the switches parse the packets and install the flow rules in the capacity-limited flow tables. The attacker can exploit the vulnerability of this reactive packet processing mechanism by flooding malicious packets to the switches. The header fields of these packets are filled with deliberately forged values that it is almost impossible for them to be matched by any existing flow entries in the switches. After that, numerous table-misses are triggered and a large number of `packet-in` messages are flooded to the controller, making the entire SDN system suffer from resource exhaustion. In this adversary model, three levels of SDN resources are compromised:

- **OpenFlow switches**: On the one hand, these malicious packets would use up the buffer memory of the switches, amplifying traffic to the control plane and degrading the

TABLE I
PACKET-IN HANDLER FUNCTION AND FORWARDING GRANULARITY IN THE MAINSTREAM SDN CONTROLLERS

Controller Platform	`Packet-in` Handler Function	Forwarding Granularity
OpenDaylight	public void processPacketInMessage (final PacketInMessage packetInMessage)	flowWriterService.addMacToMacFlow (sourceMac, destMac, nodeConnectorRef)
ONOS	public void process(PacketContext context)	setUpConnectivity(context, srcId, dstId)
Floodlight	public Command processPacketInMessage (IOFSwitch sw, OFPacketIn pi, IRoutingDecision decision, FloodlightContext cntx)	routingEngineService.getPath(srcSw, srcPort, dstAp.getNodeId(), dstAp.getPortId())
Ryu	def _packet_in_handler(self, ev)	None
Nox	Disposition trackhost_pktin handle_pkt_in(const Event& e)	routing→setup_route(flow, route, inport, outport, flow_timeout, bufs, check_nat)
Pox	def _handle_PacketIn (self, event)	_get_path(self, dst_sw, event.port, last_port)

performance of packet forwarding. On the other hand, even if the controller has the capability to handle these `packet-in` messages, so many flow rules would exceed the limited flow table capacity in the switches that new flow entries have to be dropped. Although OpenFlow 1.4 and above [10] allow switch to automatically eliminate some entries to make space for newer entries, this may make things worse since it provides an opportunity for the attacker to commit the flow table overflow attack [5]. By exploiting the vulnerability of the existing eviction mechanism, flow entries used by benign users can be replaced easily by aggressive useless fake flows from the attacker, which would lead to more table-misses for benign users and seriously degrade the performance of the entire network system.

- **Control channel**: According to the `packet-in` mechanism, each message would contain the entire packet when the buffer memory is full, which is then an amplification attack. Meanwhile, in response to these `packet-in` messages, control messages like `flow-mod` and `packet-out` messages are issued by the controller, making the control channel even more crowded.
- **Controller**: The controller has to apply some expensive operations to all `packet-in` messages, including calculating the routing path, encapsulating the flow rules into the control messages, and issuing the control messages to the data plane. Thus, it can consume a lot of computation and memory resources.

B. State-of-the-art Defense Approaches

As discussed above, the data-to-control plane saturation attack targets at the SDN infrastructure, which makes it one of the most crucial problems that must be solved for real SDN deployment. To mitigate this saturation attack, several approaches emerge in the latest academia researches. However, each of them has some prominent problems, which make it challenging to be applied and deployed in real world scenarios.

AVANT-GUARD [2] is the first defense system against this saturation attack. It extends the architecture of OpenFlow switch with a TCP proxy to mitigate TCP-based saturation attacks. However, it does not apply to other protocols (e.g. UDP, ICMP). Besides, it needs to modify the design of the data plane, thus it is difficult to be deployed.

FloodGuard [3] also proposes a defense framework to defend against this saturation attack. It introduces two techniques, *proactive flow rule analyzer* to preserve network policy enforcement and *packet migration* to protect the controller from being overloaded. However, it is impossible to install all possible flow rules into the switches with proactive flow rule analyzer because of complicated applications and controller states. In addition, extra devices (*Data Plane Cache*) have to be deployed for packet migration. Moreover, it focuses on the protection of the controller while neglecting the data plane.

FloodDefender [4] applies three techniques to protect the victim switch, *table-miss engineering* to offload the table-miss traffic to neighbor switches, *packet filtering* to filter the one-packet flows from triggering `packet-in`, and *flow table management* to eliminate useless flow rules periodically. Four corresponding modules standing between the controller platform and other apps are designed to protect both the control plane and data plane resources. However, on one hand, all the table-miss traffic is sent to control plane, regardless of whether it is forgery or not. On the other hand, all the modules introduced incur a large number of control channel traffic themselves. As a result, the effects of protection are compromised inevitably.

III. OBSERVATION AND MOTIVATION

From the analysis above, all state-of-the-art defense approaches expose some prominent issues, which motivates us to revisit the problem of saturation attack. We attribute the root cause of this saturation attack to the *reactive* packet processing mechanism introduced by SDN. Although we could mitigate this saturation attack by limiting reactive flows and pre-installing rules for all expected traffic, this comes at the expense of fine-grained control, visibility, and flexibility in traffic management. Tackling this attack without compromising the advantages of SDN is really crucial and challenging.

We re-examine this problem from the origin of the saturation attack: table-misses and `packet-in` messages. These numerous `packet-in` requests not only overflow the buffer memory and flow tables in the switches, but also occupy the bandwidth of the control channel and exhaust the computation and memory resources in the controller. Therefore, the

fundamental way to prevent this saturation attack is to *reduce the table-misses*, that is, to reduce the probability that new incoming flows fail to match existing flow entries. The new coming flows could be categorized into the following two types: normal traffic, with reliable source addresses and normal behaviors; and malicious traffic, whose packet headers are deliberately forged in order to increase the number of table-misses and the difficulty for *statistical accountability*.

Based on the observations above and inspired by the set of RFCs on source address validation (SAVI) [11], [12], [13], [14], [15] in traditional networks, we introduce *source address validation* to discard the forged packets at the entrance of SDN-enabled network. However, where and how to enforce source address validation in the complicated SDN environment gracefully remain an unresolved challenge. Although the idea (source address validation) exists long in legacy networks, to apply it harmoniously with unique SDN characteristics is non-trivial. We solve several challenges when we apply our SAVI-based techniques. Furthermore, even if the source addresses of the incoming traffic are trustworthy, packets with origin source address could also be harnessed by the attacker to commit malicious attacks. To tackle this issue, we introduce *stateful packet supervision*. However, how to conduct dynamic countermeasures with traffic features and network resource usages remains a new obstacle. We design various effective approaches to track the historic behaviors of packets passing the data plane, to achieve network service differentiation and to measure the network resource usage. With source address validation and stateful packet supervision, all the traffic is validated and monitored, which prevents malicious traffic from compromising the network system.

IV. System Design

This section first gives the overall architecture of FloodShield system and then details the design of two core modules.

A. System Architecture

FloodShield could be implemented on the existing SDN network operation systems (e.g. OpenDaylight [16], ONOS [17], Floodlight [18], RYU [19] and etc.). It introduces *Source Address Validation* Module and *Stateful Packet Supervision* Module to the controller platform. Source Address Validation Module constitutes the first barrier to the data-to-control plane saturation attack. It validates the source addresses of the incoming traffic and filters the forged packets[2] directly in the data plane. Based on it, Stateful Packet Supervision Module monitors the packet states of each real address and performs network service differentiation according to the evaluation scores and network resource usage. Moreover, to make the two modules work better with different network environments, all the parameters could be configured and adjusted through a REST API.

[2]In the following part of this paper, *forged packets* refers to packet with forged source address.

As shown in Figure 2[3], Source Address Validation Module works when a host connects to the SDN-based network. Based on the network configurations, it enables static/dynamic mode and issues *Filter Flow Rules* to the switch through which traffic enters the network. With this Module, forged packets are filtered at the ingress port of edge switch, which makes *statistical accountability* further easier. Stateful Packet Supervision Module collects traffic statistics from the data plane switches and updates evaluation scores of each user/host periodically. With the scores, different countermeasures are adopted to different users to achieve network service differentiation dynamically.

B. Source Address Validation Module

As indicated in Section III, the attacker tends to commit the saturation attack with forged source address to hide his intents and protect himself. However, the addresses zombie hosts use are allocated by the network and thus could be prior determined, especially in SDN-based data centers, enterprise networks and campus networks. Based on this insight, we introduce the *source address validation* mechanism into the defense of this saturation attack. However, the position and the method to enforce the source address validation remain unsolved challenges. To solve this, we exploit the characteristics of the flow table and propose an effective coordination mechanism with the smartness of the controller and the wire-speed performance of the data plane, filtering the forged packets directly in the data plane. All designs conform to the OpenFlow policy and need no additional devices, nor without any requirement for the specialized SAVI-enabled devices.

1) Source Address Validation Enforcement Position: In theory, we have to enforce the source address validation at different levels of the SDN-enabled network, i.e., on every OpenFlow switch in SDN. However, this leads to intricate switch management and coordination mechanism as well as considerable resource consumption across the network. To tackle this, we resort the unique global network management of SDN and prove that it is sufficient to ensure the validation of packets if the mechanism is enforced at the switch ports connected with end hosts or servers separately. First, we introduce the following two hypotheses:

Assumption 4.1: (**Packet Generation**) In the SDN data plane, all packets come from the hosts or servers connected with the edge switches except for the control messages.

Assumption 4.2: (**Packet Delivery**) In the SDN data plane, the function of the switch is either transferring packets from one port to another (modification to the packets is allowed) or simply dropping it.

Furthermore, different switches connect to different devices in the data plane. We classify the ports of the switches into the following three categories:

- **Switch Port** refers to the port connected with other switches.
- **Host Port** is the port connected with end hosts or servers.

[3]In this figure, *Reliable Traffic* refers to traffic with reliable source address.

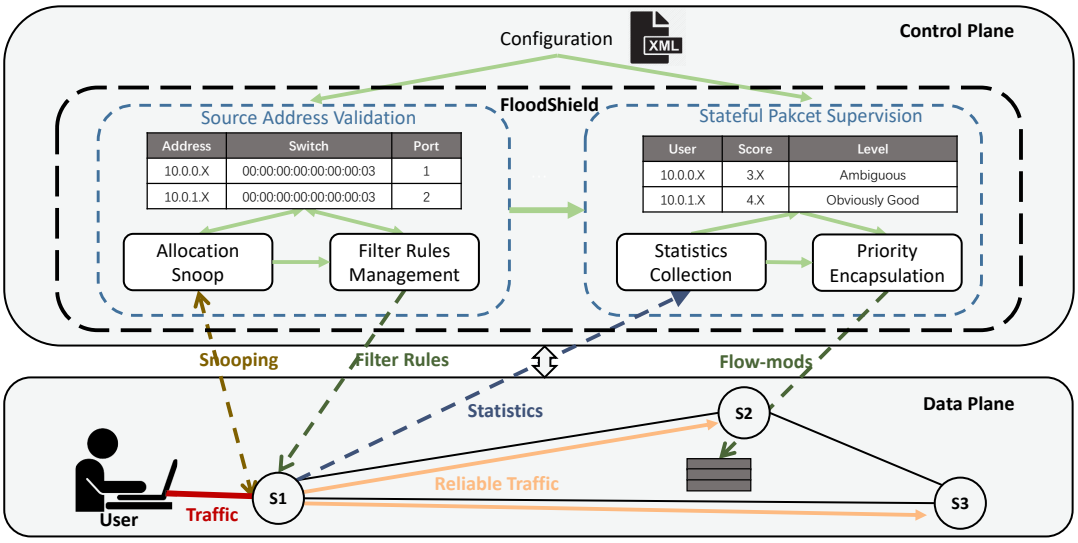


Fig. 2. Conceptual Architecture

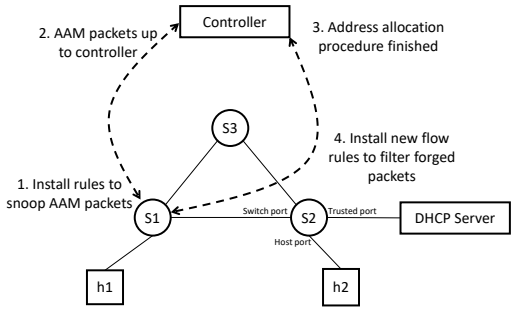


Fig. 3. SAVI on Dynamic Address Allocation

- **Trusted Port** refers to the port connected with DHCP Servers or other trustworthy servers.

Based on the two assumptions and the three port categories above, we are able to deduce that all the network traffic in the data plane is generated either from *Host Ports* or *Trusted Ports*, and is delivered between *Switch Ports*. Packets from *Host Ports* are not trustworthy while those from *Trusted Ports* are trustworthy. Therefore, as long as source address validation is performed on the *Host Ports*, the traffic in the data plane is validated and trustworthy.

2) Source Address Validation Enforcement Method: As discussed in Section IV-B1, validation at the *Host Ports* is sufficient to ensure the true origin of the traffic entering the SDN-based network. We maintain a global *Binding Table* at the controller, and each binding entry binds a source address to a switch port, in the format of $\langle Address, Switch, Port \rangle$, that is, the entry point of the packets with source *Address* should be *Port* of *Switch*; otherwise, it is a forged packet.

A host needs to obtain an address to access the SDN-based network first. However, real SDN scenarios may face numerous different host configuration protocols (static/dynamic), and network administrators may have different security and configuration requirements. Here comes our *first challenge*, how to obtain network address configurations and establish the Binding Table in the complicated network environment. To this end, we design an interface which could receive the parameters from network configurations firstly. These parameters represent whether the address configuration is static or dynamic, as well as the information to establish the corresponding Binding Table.

- For *dynamic* address allocation protocol, as illustrated in Figure 3, in order to snoop address assignment mechanism (AAM) procedure, the controller is supposed to install the snooping flow rules on *Host Ports* and *Trusted Ports* as soon as the switches connect to the controller. Then all the AAM packets go up to the controller as `packet-in` messages and the controller can monitor the whole procedure of AAM. We maintain a state machine for each AAM. Each IP address has a state, whose transition is triggered by AAM packets. When the state changes to BIND, a binding entry is set for this address in the Binding Table; when the state changes to NO_BIND, the address is removed from the table. This may become a new vulnerability when the attacker sends forged AAM packets to commit the data-to-control plane saturation attack. To solve this, we use *meter* element in OpenFlow switches to rate-limit AAM packets and we also rate-limit AAM packets in DHCP servers, since AAM traffic is low-rate under normal circumstances.
- For *static* address allocation protocol, the Binding Table could be directly obtained from the network configurations.

Based on the Binding Table, here comes the *second challenge*, how to enforce source address validation in SDN architecture. The simplest way to enforce source address validation is to do it on the controller. The controller verifies the first packet of each new flow. There are two ways to deal with a forged packet. First, ignore it. The main drawback is that

TABLE II
THE STRUCTURE OF TABLE 0 (FILTER FLOW TABLE) IN THE DATA PLANE

Group Name	Match Field	Priority	Instruction
Host Ports	in port=hport, mac src=given mac src, ipv4, ip src=given ip src	2	jump to table 1
Host Ports	in port=hport, mac src=given mac src, arp	2	jump to table 1
Host Ports's filter	in port=hport, ipv4	1	drop
Host Ports's filter	in port=hport, arp	1	drop
Switch Ports	in port=sport	1	jump to table 1
Trusted Ports	in port=tport	1	jump to table 1

the attacker could repeatedly send the forged packets to overwhelm the controller's computation resources. Second, passively install a drop rule matching the new flow into the switch. However, both the computation resources on the controller and the flow table resources on the switch are exhausted when the attacker floods packets with random source addresses. To address the problem above, Source Address Validation Module *explicitly* installs *Filter Flow Rules* in switches to filter the unbound source addresses. In this way, forged packets are verified and dropped in the data plane directly, without consuming flow table resources or computation resources on the controller.

Existing OpenFlow switches only support *match-action* paradigm, that is, if a packet matches an existing flow entry in the local flow tables, the switch would conduct the corresponding actions to process the packet. However, source address validation follows an *unmatch-action* paradigm, that is, filter actions are conducted only if the source address *does not* match the specific port claimed in the Binding Table. How to design the *Filter Flow Rules* to express our *unmatch-action* logic gracefully becomes the *third challenge*. One way is to block unbound source addresses, leaving the unmatched (legitimate) packets to be processed by the rules of other applications. However, there are too many unbound addresses, which would undoubtedly exceed the limited flow table size. Another way is to combine the rules of Source Address Validation Module with other applications. For example, for each bound source address, generate N *two-dimension* rules, where N is the number of flow rules of other applications (e.g. Forwarding). However, this method brings about the explosion of flow rules. In order to prevent the flow table explosion problem and make the flow table separate and independent, we resort to the *multi-table pipeline* of OpenFlow. The *Filter Flow Rules* are installed in table 0, while the flow rules from other applications in the controller are installed in latter flow tables. As for the detailed implementation of the *Filter Flow Rules*, inspired by the algorithm policy enforcement in Maple [20], we gracefully encode negation logic using wildcard rules with lower priorities. Filtering is only done on *Host Ports*, so the packets coming from *Switch Ports* and *Trusted Ports* are forwarded directly to latter flow tables. The structure of *Filter Flow Rules* consists of 6 flow rule groups, as shown in Table II. The first and second flow rule groups let pass all the packets with given IPv4 source address and given Ethernet source address, while the third and fourth flow rule groups drop all spoofed packets with forged IPv4 source address or forged Ethernet source address. Finally, in the fifth and sixth rule groups, packets from *Switch Ports* and *Trusted Ports* are directly forwarded to the non-filter flow tables.

C. Stateful Packet Supervision Module

Although forged packets have been filtered by the source address validation, packets with real source addresses could also be harnessed to conduct malicious attacks. Therefore, we introduce *stateful packet supervision* to perform address-based *statistical accountability*. Our basic idea is simple, flows with different source addresses should be distinguished by their traffic features, further to achieve differentiated services for different users *dynamically*[4]. However, we encounter two major challenges in turning this high-level idea into a concrete defense mechanism. 1) how to develop behavior-based evaluation criterion to distinguish flows precisely and lightly. 2) how to achieve differentiated services to different flows dynamically and effectively. The following two submodules are designed to deal with these two challenges.

1) Precise Behavior Evaluation: In order to perform the behavior-based evaluation, we first have to determine what traffic features to collect. In this unique saturation attack, `packet-in` rate is the dominant factor to influence the attack effect. To trigger more `packet-in` messages with a lower cost, the attacker tends to commit the attack with *short-flows*, since in this way he/she could gain the aim of almost per packet per `packet-in` triggering. Further, with the increase of new flow rate, the `packet-in` rate increases and the attack effects become more obvious. Therefore, we make the observation that the fundamental differences between normal traffic and attack traffic under this unique saturation attack are the *frequency of new flows* and the *packet number per flow*, which is a little different from traditional DoS attacks.

To collect these two metrics, a light and precise method is in urgent need. As each new flow would trigger a `packet-in` message to the controller under normal circumstances, passive monitoring, classifying, and counting on the controller are sufficient to obtain the frequency of new flows. However, the number of packets per flow is not directly visible to the controller. We resort to the *pull* mechanism of OpenFlow to tackle this issue. *Statistic messages* are issued to the switch periodically (every T seconds) to query the *snapshot* information (the *counter* field of the existing flow entries) the controller needs. Fortunately, the statistic service is integrated

[4]Benefit from source address validation, in this paper, we use *source address* and *user* interchangeably.

TABLE III
BEHAVIOR-BASED NETWORK SERVICE DIFFERENTIATION

Evaluation Level	Countermeasure	Importance
Obviously Malicious	drop/rate-limit in the ingress switch for a period	none/low
Ambiguous	probability acceptance	low
Obviously Good	deal all	high

into the network operation system as a basic service in most of the popular controllers (e.g. ODL, ONOS, Floodlight) so that we could invoke the service and classify the flow entries according to their source addresses directly.

With these traffic features, an evaluation criterion is developed to distinguish the behavior of different users. As discussed above, the frequency of new flows and the statistical properties of packet number per flow are two basic differences between the benign traffic and the attack traffic. For brevity, considering only the past period (T seconds), we denote the number of `packet-in` messages as pi_i, the ratio of good flow entries pulled from the switch as cr_i, where

$$cr_i = \frac{\text{flow entries}(\text{counter} > t \wedge \text{source address} == given)}{\text{flow entries}(\text{source address} == given)} \quad (1)$$

t is a constant parameter used to distinguish good flow entries from bad ones, which could be set as different values for different hosts/servers (it is normally set as an integer smaller than 10, which depends on the traffic features as we demonstrated in Section V-D). We denote the score associated with pi_i and cr_i as w_{in}. Obviously, w_{in} has a negative correlation with pi_i and a positive correlation with cr_i, since the attacker is inclined to attack with short frequent new flows. We simply use two linear correlations and add them up to express the logic above. In this way, a behavior evaluation criterion and a corresponding evaluation score are developed for each user on the controller.

2) Dynamic Service Differentiation: With the behavior-based evaluation scores, how to achieve differentiated services for different users dynamically becomes our second challenge. In some cases, an attacker may behave correctly firstly, gets classified as benign user, and then changes his/her behavior. Converse case may also emerge. To resolve this problem, we adopt the exponential weighted moving average (EWMA) and take the historic information into the evaluation of a user's behavior. The evaluation score w_i at this point obeys the EWMA paradigm: $w_i = (1-\alpha)w_i + \alpha w_{in}$, where α is a value between 0 and 1, w_in denotes the score in this period. With the attacker's behavior getting malicious, his/her evaluation score will inevitably goes down. The convergence speed depends on the α in EWMA. The bigger α is, the faster the evaluation score converges. The benefits of dynamic evaluation scores also extent to the network service differentiation. Diverse strategies are adopted to different users to achieve network service differentiation dynamically. We divide the region of scores into three levels(*Obviously Malicious*, *Ambiguous* and *Obviously Good*) with two thresholds (th_h, th_l), which are set by the network administrators based on the deployment scenario and training data set[5]. *Obviously Malicious* denotes users whose score (w_i) is lower than th_l while *Obviously Good* denotes those with a score higher than th_h, and the remaining users, with a score between two thresholds are regarded as *Ambiguous*, which means we could not determine whether it is a good flow or a bad one directly.

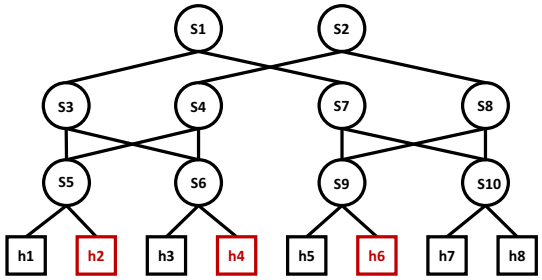


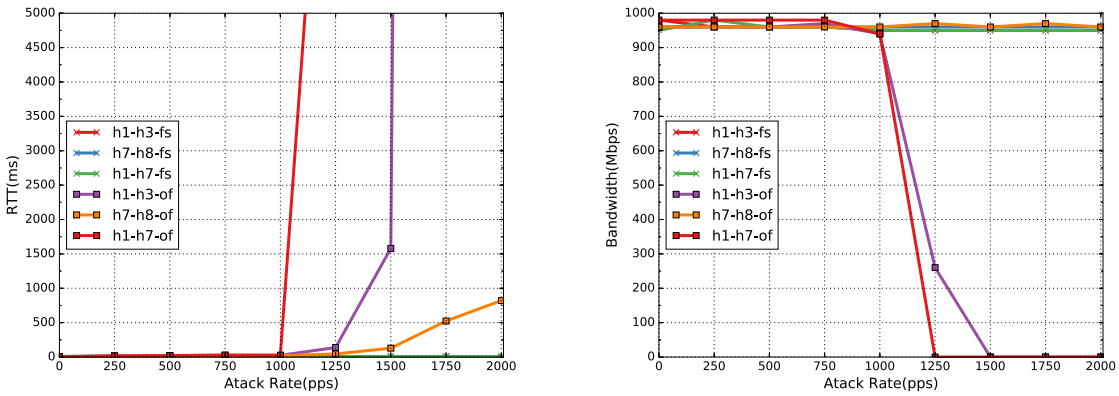
Fig. 4. Experiment Topology

Once the controller receives new flows from *Obviously Malicious* users, a *drop* or *rate-limit* flow rule will be directly issued to the ingress switch, dropping or rate-limiting packets from this source address for a period of time. We take this seemingly brute measure because the low threshold th_l is hard to be reached for normal traffic with our methodology.

Obviously Good level follows the similar law, since its threshold is hard to be reached for malicious traffic normally. We let the new flows of this level be processed by controller as usual. Considering the occasion that the attacker may commit flow table overflow attacks [5], we assign different *importance* to different `flow-mod` messages to protect the flow table in the data plane. Flows come from *Ambiguous* users are assigned with lower *importance* while those from *Obviously Good* users are assigned with higher *importance*.

As for *Ambiguous* flows, we take a *probability acceptance algorithm* based on the controller resource usage. When the controller is busy, the probability value is low, and thus only a smaller number of `packet-in` messages are accepted and processed. By contrast, nearly all `packet-in` messages are dealt when the system is relatively idle. Supposing the SDN controller is able to handle N `packet-in` messages per seconds, and the controller receives N_R `packet-in` messages at this point, we use $\frac{N_R}{N}$ to denote the network resource usage. Obviously, the probability value has a negative correlation with the network resource usage, and we take a simple linear correlation to implement it. All above-mentioned

[5]The rationality and methodology for the *Two thresholds/Three levels* design are illustrated in Section V-D, which can effectively reduce the loss caused by false-positive and false-negative rate.



(a) Round-trip Time

(b) Available Bandwidth

Fig. 5. End-to-End Effect

strategies are summarized as Table III.

V. Implementation and Evaluation

In this section, we introduce the implementations of FloodShield and related works, and evaluate the performance and overheads of our framework.

A. Implementation and Experiment Setup

To verify the effects of FloodShield, we have implemented a prototype on top of Floodlight controller by integrating *Source Address Validation* Module and *Stateful Packet Supervision* Module to the controller platform. All the implementation conforms to the OpenFlow policy and needs no additional devices nor hardware re-design of the SDN infrastructure, which can be generalized to the other SDN controllers easily. The OpenFlow-enabled network is emulated with Mininet [21] and OpenvSwitch [22].

We emulate a simple fat tree topology (Figure 4) based on Floodlight and Mininet, running on two separate and identical servers connected by a conventional layer-2 hardware switch. The hardware settings of the servers are two Intel Xeon-2620 CPUs, 64GB RAM and a 1 GbE NIC. In order to better simulate the real scenario, all the link bandwidths in the topology are set to 1 Gbps, and the flow table size of each switch is set as 2000, making it analogous to the hardware Pica8 P-3290 switches [23].

To compare FloodShield with previous works, we conduct our experiments in the following four scenarios, to show the advantages of our FloodShield: (1) an OpenFlow network without protecting system (*of*), (2) an OpenFlow network with FloodGuard [3] (*fg*), (3) an OpenFlow network with FloodDefender [4] (*fd*), (4) an OpenFlow network with our FloodShield (*fs*). Unfortunately, due to copyright reasons, we are not able to obtain the origin source codes of FloodGuard and FloodDefender, so we reproduce all the logic of FloodGuard and FloodDefender by ourselves. In the following part of this section, we use the abbreviation, *of, fg, fd, fs*, to represent the four system above.

B. End-to-End Effect

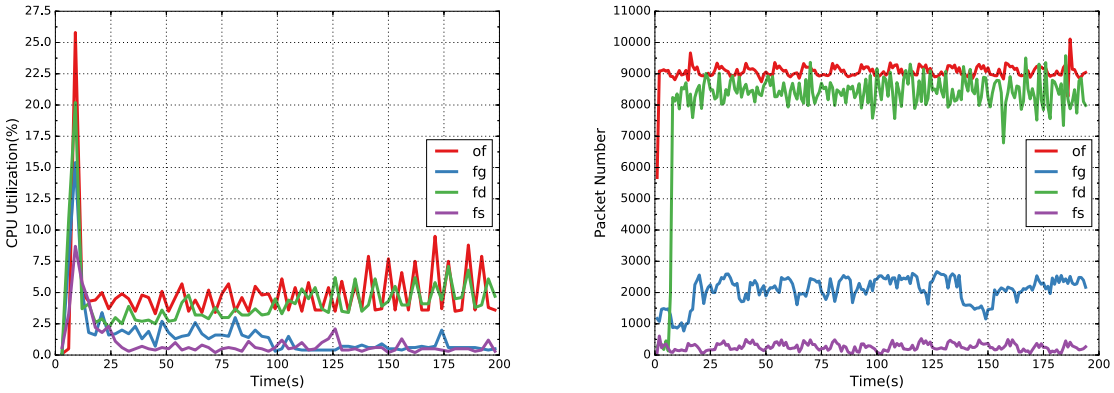
To evaluate the end-to-end protective effect of FloodShield for the SDN infrastructure, we measure *Round-Trip Time (RTT)* and *available bandwidth* between hosts as representative metrics.

We let h2, h4, and h6 be the attack hosts, generating different rates of flooding traffic with *Tcpreplay*[6]. The traffic is extracted from an online trace dataset, *DARPA 2009 malware-DDoS attack-20091104*[7] [24]. And simultaneously we make h7 access h8, h1 access h3, and h1 access h7 with *ping* to evaluate the RTTs and *iperf* to evaluate the available bandwidths. h7-h8 represents communication between two hosts via edge layer switches, h1-h3 represents communication via edge layer and aggregation layer switches, and h1-h7 represents communication via all three layer switches. Each experiment is ran for 10 times and the average value is presented in Figures and Tables. Because FloodGuard, FloodDefender and our FloodShield show similar results on RTTs and available bandwidths, for brevity, we only show the results of FloodShield to demonstrate the end-to-end protective effect.

As Figure 5(a) shows, with native OpenFlow, the RTTs become extremely large when the attack rate is above 1000 packets per second (pps) from the three attackers, while with FloodShield, the RTTs are almost unchanged. Figure 5(b) demonstrates the similar results on the available bandwidths. Bandwidths under native OpenFlow go down quickly at 1000 pps while FloodShield's keep nearly unchanged. That is because OVS could afford about 1200 pps flooding rate under normal circumstances. Besides, FloodShield has neglectable impact on the RTTs or available bandwidths in the data plane when there is no saturation attack. This has been verified in Figure 5 when attack rate is 0 pps. The experimental results on RTTs and bandwidths illustrate that the FloodShield provides effective end-to-end protection for the SDN infrastructure.

[6]http://tcpreplay.appneta.com

[7]Since there are a large number of elephant flows in this trace and they are not friendly to the attacker, so we only extract the mice flows from the dataset.



(a) CPU Utilization of Controller

(b) Control Channel Utilization

Fig. 6. Resource Utilization

TABLE IV
AVERAGE FLOW ENTRY NUMBER

	Normal Users	Attackers	Overheads	All
OpenFlow	535.30(27%)	1423.14(71%)	0(0%)	1959.21(98%)
FloodGuard	389.77(19%)	508.38(25%)	2.75(0.3%)	900.77(45%)
FloodDefender	492.09(25%)	250.37(13%)	571.21(28%)	1313.11(66%)
FloodShield	679.17(34%)	97.70(5%)	5.32(0.6%)	799.87(39%)

C. Resource Utilization

The saturation attacks would exhaust the resources of the SDN infrastructure, most notably the CPU of the controller, controller-switch channel and flow table space. To demonstrate the advantages of FloodShield, we conduct the experiments compared with the three system above.

We let h2, h4, and h6 be the attack hosts replaying the real trace from *DARPA 2009 malware-DDoS attack-20091104* with 500 pps (the same rate as FloodDefender), and simultaneously all the hosts access each other under normal circumstances with different connections (by replaying traffic collected from campus network of Tsinghua University). We record the number of control messages and the CPU utilization of controller for statistics and analyses. We also record the the number of flow entries allocated to normal users and attackers in victim switch during the past 30 seconds as the metric of flow table utilization.

Controller CPU Utilization: The CPU utilization of the controller is shown in Figure 6(a). We find that FloodShield occupies much less CPU than native OpenFlow as well as FloodDefender. As FloodGuard highlights, its main goal is to protect the controller, and it achieves a similar protective effect for controller as our FloodShield. Meanwhile, as we argue, FloodDefender puts too much burden on the controller, and the evaluation results also demonstrate our argument. FloodShield filters the forged packets and supervises the packets with real source addresses, effectively preventing malicious packets from compromising the SDN controller.

Control Channel Utilization: We limit the scope of control messages to `packet-in`, `packet-out`, `flow-mod` and `flow-removed`, since the number of others like `LLDP` messages is similar in the four systems. Figure 6(b) shows that the number of control messages of four systems. With proactive flow rule installation and rate control, FloodGuard reduces the control channel utilization greatly. FloodDefender reduces the `flow-mod` messages for the malicious `packet-in` requests, meanwhile, it needs new `flow-mod` messages to conduct the monitoring rules flushing, as a result, its control channel utilization is similar to the native OpenFlow. Compared with the two protective systems above, FloodShield manifests a much better protective effect. The forged packets are directly dropped in the data plane, without any control messages any more. The rest of the packets are supervised, which provides a more comprehensive protective effect for the control channel.

Flow Table Utilization: The results of flow table utilization are listed in Table IV. FloodShield occupies much less flow entries (39%) compared with OpenFlow (98%), FloodGuard (45%) and FloodDefender (66%), and provides better services for normal users while suppressing the attackers effectively. FloodGuard uses rate control to protect the controller and switches, thus flow entries of both normal users and attackers decrease greatly (45%/98%). FloodDefender resorts to a two-phase filter and flushes monitoring rules and cache regions periodically, as a result, it reduces the flow table used by attackers (13%) and maintains a reasonable number of flow table for normal users (25%). However, this benefit comes at the expense of monitoring flow rules in the switches (28%). FloodShield protects the flow table more thoroughly. On the one hand, it filters the forged packets directly in the data plane. On the other hand, it supervises the network states of data plane traffic, making it hard for the attackers to occupy the

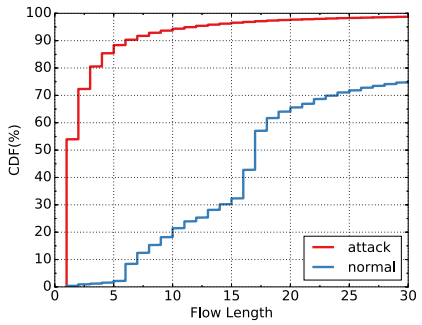
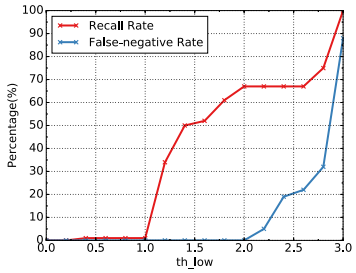
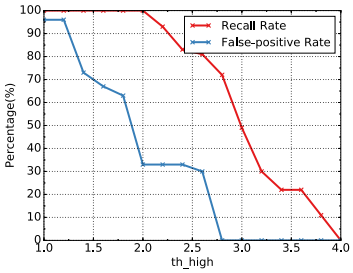


Fig. 7. CDF of Flow Length



(a) Relationship of Recall Rate/False-negative Rate and th_l



(b) Relationship of Recall Rate/False-negative Rate and th_h

Fig. 8. Recall Rate and False-positive/False-negative Rate

flow entries.

D. Attack Identification and Parameter Analysis

In this part, we revisit Stateful Packet Supervision Module and dig into more details. We give some suggestions on how parameters should be set with the analysis of difference traces, and elaborate the recall rate and false-positive/false-negative rate of FloodShield.

Figure 7 shows the distribution of flow length (packet number per flow) for attack traffic and benign traffic (the same traces as Section V-C). As we can see, attack traffic inclines to have more short flows than benign traffic. When t is set between 2 and 5, we are able to obtain the most significant discrimination between the attack traffic and benign one. Thus, t is recommended to bet set in this range for this traffic patterns. Although different scenarios may have unique traffic features, we believe that there must be significant distinction for their flow length distribution, otherwise the attacker has to spend several order of expense to get the same attack impact, which is further not cost-effective and lead to easier detectable.

With t set as the recommended constant, we calculate and record the evaluation score for normal users and attackers. With different th_l and th_h, we are able to get different recall rate and false-positive/false-negative rate. Generally speaking, the smaller th_l is, the lower false-negative rate and recall rate would be; the larger th_h is, the lower false-positive rate and recall rate would be. Figure 8 demonstrates our evaluation results. As we can see, when th_l is set as 2, we could identify 67% attack traffic (*Obviously Malicious*) with almost 0 false-negative rate. Accordingly, we are able to distinguish 72% benign traffic (*Obviously Good*) with nearly 0 false-positive rate when th_h is set as 2.8. We suggest that th_l and th_h follow this principle, and both the undistinguished attack traffic and benign traffic fall into the *Ambiguous* level, which would contend for the limited network resources with the probability acceptance algorithm. In addition, we can also see how *sensitive* FloodShield is to th_l and th_h from Figure 8. With a small oscillation near the thresholds, the recall rate and false-positive/false-negative rate manifest fairly robust.

E. Overhead

In this section we show our evaluation about the overheads of FloodShield. First, to filter the forged packets in the data plane, FloodShield requires *Filter Flow Rules* in the edge switch. In particular, the number of *Filter Flow Rules* is proportional to the the number of *Host Ports* in the edge switch, with a coefficient falling between 2 and 4. Standard switch has no more than 52 ports [25], even if no port is used as *Switch Port* or *Trust Port*, the total number of *Filter Flow Rule* is less than 200, which only occupies at most 10% of flow table (2000 flow table capacity) for a commodity hardware switch in the worst case. Therefore, we argue that the overheads for *Filter Flow Rules* is negligible.

Second, Stateful Packet Supervision Module seems to be time-consuming components, since each packet may trigger the state transition. We remove Source Address Validation Module and only reserve the Stateful Packet Supervision Module on the controller platform. As shown in Figure 9, the CPU utilization of FloodShield is similar to the native Floodlight, and it incurs less than 3% overheads in the worst case. In particular, when the `packet-in` rate reach 400 pps, the CPU utilization of FloodShield increases much slower. This is because the Stateful Packet Supervision has *self protection* functions to some degree. When the new incoming `Packet-in` packets reach a high rate (to an intolerable degree to network resources), the module would issue some rate-limit (or drop) flow rules to the data plane, reducing the rates of new `packet-in` messages.

Third, we also evaluate the delay resulted from the additional two modules under normal circumstance. For each flow, the delay of first packet includes two aspects: the two-stage switch delivery (for Source Address Validation) and the stateful controller processing (for Stateful Packet Supervision). For other packets, they only go through delay of the two-stage switch delivery. As shown in Figure 10, compared with native Floodlight, delays for these two kinds of packets are negligible.

VI. Discussion

Distributed controllers: One argument is that the saturation attack may not take effect when distributed controllers are adopted, since this means bigger throughput and stronger processing capability in the control plane. However, we believe that even the distributed approach is taken, the attack could still paralyze the SDN infrastructure without proper treatment. As the *barrel principle* indicates, with a powerful controller,

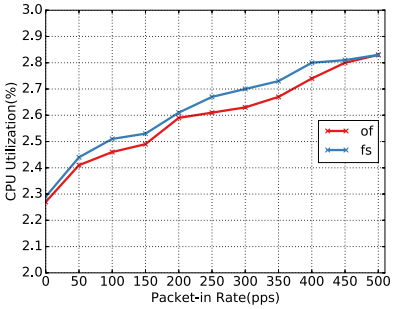
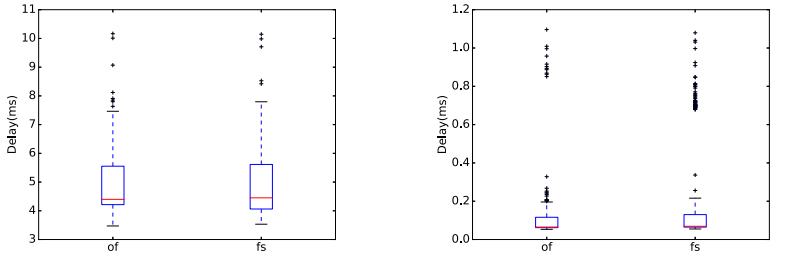


Fig. 9. CPU Overhead



(a) Delay of First Packets in Each Flow

(b) Delay of Following Packets in Each Flow

Fig. 10. Delay of the First Packet and Following Packet in Each Flow

control channel or data plane switch would be the bottleneck under this circumstance. Thus, the bottleneck transfers from one point (controller) to another (control channel or data plane), instead of being mitigated.

Volatile address problem: In some scenarios, an IP address may be reassigned to another host frequently. Fortunately, the source address validation process has overcome this problem. After a host releases an IP address, based on the AAM procedure, the controller will delete the corresponding table entry in the Binding Table and add the new bound pair in the table immediately. Thus the system is able to work properly even under this circumstance.

Traffic feature imitation problem: The evaluation criterion and the selected features are unknowable to the attacker under normal circumstances. Even if he could obtain these information, the cost for this overflow attack is multiplied, for he has to send multiple packets for each flow and reduce the new flow frequency to get a higher evaluation score.

Source address forgery problem: Source Address Validation puts the root of trust at the port of the edge switches, since one port corresponds to one host by default. Therefore, if the default cannot be satisfied and multiple hosts connect to one port of a edge switch, the attacker is able to forge source address and pollute the evaluation of the hosts under the same port. However, the scope of the pollutions is confined to the hosts under the same port, not the entire network.

Implications of identity hijacking attacks: FloodShield establishes the root of trust on AAM procedures or static configurations. It enforces a strong binding between a host, the network location it connects, the address in the packets it generates, which makes it resilient to identity hijacking attacks, just as discussed in Ethane [26]. Furthermore, FloodShield is much deeper than Ethane at this point. It carries out the identity binding task as *Filter Flow Rules* effectively and dexterously in the data plane while Ethane commits the task with judging logics in the control plane, leaving the control channel and control plane still suffering, just as the second challenge in Section IV-B discussed.

VII. Related Work

SDN-supported security: SDN has offered a new chance to solve some security problems that have long existed in networking. CloudWatcher [27] proposes a Security-Monitor-as-a-Service framework for cloud network. Braga et al. [28] propose a lightweight DDoS attack detection method with traffic flow features extracted from the controller. Further, Xu et al. [29] propose an adaptive detection approach leveraging limited TCAM available on all switch to balance the coverage and granularity of detection. FRESCO [30] offers a Click-inspired programming framework to facilitate the rapid design and modular composition of detection and mitigation modules. Bohatei [31] proposes a flexible and elastic DDoS defense system for ISPs. SPIFFY [32] introduces SDN to the defense of link-flooding attacks. To protect against packet spoofing, our preliminary poster, SD-SAVI [33], uses SDN to enforce source address validation, and Afek et al. [34] propose to implement several anti-spoofing methods with match-action model of SDN data plane. Some of them may have used certain techniques that are also used in our work such as source address validation, but aim at other specific problems. Our work is the first to apply these techniques to protect the SDN infrastructure.

The security of SDN itself: SDN has also introduced new security concerns. To facilitate detection and mitigation function deployment in OpenFlow-enabled networks, FRESCO [30] proposes a security application development framework which allows rapid designing and modular composition of these functionalities. FortNOX [35] provides *role-based authorization* and *security constraint enforcement* for the NOX OpenFlow controller, enabling NOX to check flow rule contradictions. Rosemary [36] introduces a micro-NOS sandboxing strategy to the OpenFlow controller to safeguard the control layer from errant operation performed by network applications. TopoGuard [37] introduces two Network Topology Poisoning Attacks (*Host Location Hijacking Attack* and *Link Fabrication Attack*) against the SDN/OpenFlow Topology Management Service and proposes automatic mitigation approaches to prevent the attacks. DELTA [38] presents a fuzzing-based penetration testing framework to find unknown attacks in SDN controllers. CONGUARD [39] introduces the *state manipulation attack* by exploiting harmful race conditions in the SDN controller. SECUREBINDER [40] proposes the Persona Hijacking attack to break the bindings of all layers of the SDN networking stack, and designs a mitigation solution building

upon IEEE 802.1x as a root of trust. These approaches target at specific problems which are different from ours.

VIII. Conclusion and Future Works

In this paper, we propose FloodShield, a comprehensive, deployable and lightweight SDN defense framework, to mitigate the data-to-control plane saturation attack against the SDN infrastructure. Based on analysis on the vulnerability of SDN under this saturation attack, we design FloodShield which consists of two components: source address validation and stateful packet supervision. Evaluations demonstrate that, compared with state-of-the-art defense frameworks, FloodShield provides effective protection for the data plane, control channel, and controller with less resource consumption and negligible overheads.

Securing the network infrastructure is crucial to the promotion and adoption of SDN. In our future works, we will try to solve the challenge that how to enforce source address validation in the gateway where SDN-based network connects to the traditional network, for this scenario is much more complicated than the pure SDN-based network and has more practical significance. Furthermore, we will deploy our proposal in the real world, which would provide greater insights.

Acknowledgement

This work is supported by National Key R&D Program of China (2017YFB0801701), the National Science Foundation of China (No.61472213) and CERNET Innovation Project (NGII20160123). We are grateful to anonymous reviewers for their valuable and constructive comments.

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," *SIGCOMM*, vol. 38, no. 2, pp. 69–74, 2008.

[2] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *CCS*. ACM, 2013, pp. 413–424.

[3] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in *DSN*. IEEE, 2015, pp. 239–250.

[4] S. Gao, Z. Peng, B. Xiao, A. Hu, and K. Ren, "Flooddefender: protecting data and control plane resources under sdn-aimed dos attacks," in *INFOCOM*, 2017, pp. 1–9.

[5] M. Zhang, J. Bi, J. Bai, Z. Dong, Y. Li, and Z. Li, "Ftguard: A priority-aware strategy against the flow table overflow attack in sdn," in *SIGCOMM*. ACM, 2017.

[6] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simplefying middlebox policy enforcement using sdn," *SIGCOMM*, vol. 43, no. 4, pp. 27–38, 2013.

[7] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *SOSR*. ACM, 2016, p. 6.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM*, vol. 43, no. 4, pp. 3–14, 2013.

[9] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *SIGCOMM*, vol. 43, no. 4. ACM, 2013, pp. 15–26.

[10] "Openflow switch specification version 1.5.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf, ONF, 2014.

[11] J. Bi, K. Xu, X. Li, M. Williams, J. Wu, and G. Ren, "A source address validation architecture (sava) testbed and deployment experience (rfc 5210)," 2008.

[12] E. Levy-Abegnoli, "Fcfs savi: First-come, first-served source address validation improvement for locally assigned ipv6 addresses (rfc 6620)," 2012.

[13] J. Bi, J. Wu, M. Bagnulo, C. Vogt, and F. Baker, "Source address validation improvement (savi) framework (rfc 7039)," 2013.

[14] G. Yao, J. Wu, J. Bi, and F. Baker, "Source address validation improvement (savi) solution for dhcp (rfc 7513)," 2015.

[15] G. Yao, J. Bi, J. Halpern, and E. Levy-Abegnoli, "Source address validation improvement (savi) for mixed address assignment methods scenario (rfc 8074)," 2017.

[16] "Opendaylight," https://www.opendaylight.org/, January 2018.

[17] "Onos," http://onosproject.org/, January 2018.

[18] "Floodlight," http://www.projectfloodlight.org/floodlight/, January 2018.

[19] "Ryu," https://osrg.github.io/ryu/, January 2018.

[20] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying sdn programming using algorithmic policies," *SIGCOMM*, vol. 43, no. 4, pp. 87–98, 2013.

[21] "Mininet. rapid prototyping for software defined networks," http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/, January 2018.

[22] "Open vswitch," http://openvswitch.org/, January 2018.

[23] "Flow scalability per broadcom chipset," http://www.pica8.com/wp-content/uploads/2015/09/v2.9/html/hardware-guides/, January 2018.

[24] "Darpa scalable network monitoring (snm) program traffic," http://www.isi.edu/ant/lander, January 2018.

[25] "Understanding the different types of ethernet switches," https://blogs.cisco.com/smallbusiness/understanding-the-different-types-of-ethernet-switches, January 2018.

[26] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Mckeown, and S. Shenker, "Ethane: taking control of the enterprise." SIGCOMM, 2007, pp. 1–12.

[27] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks," in *ICNP*. IEEE, 2012, pp. 1–6.

[28] R. Braga, E. Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow," in *LCN*. IEEE, 2010, pp. 408–415.

[29] Y. Xu and Y. Liu, "Ddos attack detection under sdn context," in *INFOCOM*. IEEE, 2016, pp. 1–9.

[30] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *NDSS*, 2013.

[31] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic ddos defense." in *USENIX Security Symposium*, 2015, pp. 817–832.

[32] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *NDSS*, 2015.

[33] B. Liu, J. Bi, and Y. Zhou, "Source address validation in software defined networks," in *SIGCOMM*. ACM, 2016, pp. 595–596.

[34] Y. Afek, A. Bremler-Barr, and L. Shafir, "Network anti-spoofing with sdn data plane," in *INFOCOM*. IEEE, 2017, pp. 1–9.

[35] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *HotSDN*. ACM, 2012, pp. 121–126.

[36] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *CCS*. ACM, 2014, pp. 78–89.

[37] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures." in *NDSS*, 2015.

[38] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "Delta: A security assessment framework for software-defined networks," in *NDSS*, vol. 17, 2017.

[39] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the sdn control plane," in *USENIX Security Symposium*. USENIX Association, 2017, pp. 451–468.

[40] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *USENIX Security Symposium*. USENIX Association, 2017, pp. 415–432.