2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks

Haopei Wang
SUCCESS Lab
Texas A&M University
haopei@cse.tamu.edu

Lei Xu
SUCCESS Lab
Texas A&M University
xray2012@cse.tamu.edu

Guofei Gu
SUCCESS Lab
Texas A&M University
guofei@cse.tamu.edu

***Abstract*—This paper addresses one serious SDN-specific attack, i.e., *data-to-control plane saturation attack*, which overloads the infrastructure of SDN networks. In this attack, an attacker can produce a large amount of table-miss `packet_in` messages to consume resources in both control plane and data plane. To mitigate this security threat, we introduce an efficient, lightweight and protocol-independent defense framework for SDN networks. Our solution, called FLOODGUARD, contains two new techniques/modules: *proactive flow rule analyzer* and *packet migration*. To preserve network policy enforcement, proactive flow rule analyzer dynamically derives *proactive flow rules* by reasoning the runtime logic of the SDN/OpenFlow controller and its applications. To protect the controller from being overloaded, packet migration temporarily caches the flooding packets and submits them to the OpenFlow controller using rate limit and round-robin scheduling. We evaluate FLOODGUARD through a prototype implementation tested in both software and hardware environments. The results show that FLOODGUARD is effective with adding only minor overhead into the entire SDN/OpenFlow infrastructure.**

***Keywords*—Software-Defined Networking (SDN); Security; Denial-of-Service Attack;**

I. INTRODUCTION

Software-Defined Networking (SDN) [21] has quickly emerged as a new promising technology for future networks, and its reference implementation, OpenFlow [24], is becoming widely used in recent years [1]. SDN presents a physically distributed but logically centralized controlled networking framework. By decoupling the control plane from the data plane, SDN is designed to support fine-grained network management policies. Current OpenFlow implementations use a "southbound" protocol. When a switch receives a new flow for which there is no matching flow rules installed in the flow table (we call it a "table-miss" in this paper), the data plane will ask the control plane for actions.

The "southbound" protocol of an OpenFlow controller introduces considerable overhead. A table-miss could consume resources (e.g., CPU, memory and bandwidth) in both control plane and data plane. This leads to issues in both scalability and security. While there are many studies and solutions on the scalability issue [20], [12], [10], [32], [14], there is very little research on the even more challenging security issue. Essentially, a large number of data plane messages will flood the control plane and could exceed the throughput and processing capacities of the control plane. An attacker can exploit it by launching dedicated *denial of service attack* (or *data-to-control plane saturation attack*) that floods SDN networks [27], [29]. The attacker only needs to generate a large number of anomalous packets, with all or part of header fields of each packet are spoofed as random values. These incoming packets will trigger table-misses and send `packet_in` messages to the controller. As a result, this attack will overload the buffer memory of network devices, generate amplified traffic to occupy the data-to-control plane bandwidth, and consume the computation resource of the controller in a short time. While some existing research has already discussed this attack and presented some solutions, e.g., AvantGuard [29], they do not provide a comprehensive solution yet. For example, AvantGuard can only defeat TCP-based flooding attacks, but not others.

In this paper we study the *data-to-control plane saturation attack* in reactive controllers (e.g. POX [5] and Floodlight [6]). The impact of this attack on different controller applications is quite different. Each application in the controller consists of multiple packet-processing policies. These policies are high-level and used to generate low-level flow rules to the data plane. The applications need to analyze each `packet_in` messages, extract required information (packet header, data path, inport and etc.) and process response OpenFlow messages. Different applications have different program logic, architecture and throughput. Similarly, the impact of this data-to-control plane saturation attack on the OpenFlow infrastructure differs in target applications. For example, a load balancing application is more vulnerable than a hub application. It is because the former one needs more programming complexity to handle `packet_in` messages and respond to traffic load dynamics. The flow rules generated by the load balancing application may frequently change during the saturation attack, which means this attack could consume network resources more quickly by attacking this application.

To defend against this attack, we have two research challenges as follows:

- How to keep the major functionality of the SDN infrastructure working when the saturation attack occurs?
- How to handle the flooding traffic without sacrificing benign traffic?

For the first challenge, we propose a solution that is based on proactively placing the potential rules into the switches to

[1] In this paper, we use SDN and OpenFlow interchangeably.

978-1-4799-8629-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DSN.2015.27

guarantee the policy enforcement of the OpenFlow controller and its applications during the data-to-control plane saturation attack. Motivated by existing work [23], [11], [16], we attempt to use program analysis techniques to solve the first challenge. We define an important concept that will be used in this paper, i.e., proactive flow rules. Proactive flow rules are all data plane level (low-level) flow rules that can be generated by an application based on current controller state. The proactive flow rules represent the forwarding actions for all possible incoming packets at this moment. In this paper, we introduce a new approach that combines symbolic execution and dynamic application tracking to generate proactive flow rules. The proactive insertion of the flow rules can keep the major functionality of the network working.

However, even with the proactive insertion of the flow rules, the OpenFlow controller could still be vulnerable to the overloading problem during the data-to-control plane saturation attack. It is because most of flooding packets stay outside the logic of OpenFlow controller applications, i.e., the flooding traffic can hardly match proactive flow rules, which will actually send back and overload the OpenFlow controller. One simple solution to protect the OpenFlow controller can be dropping those packets if they cannot match any existing data plane flow rules. However, the naive drop solution inevitably scarifies some normal traffic, which may not be covered by current proactive flow rules. To solve the second challenge, we migrate all the table-miss packets to a data plane cache component, which temporarily caches the packets to protect the data plane switches and forwards them to the controller in a rate limited manner. The data plane cache is coordinated by the migration agent inside the OpenFlow controller, which provides utility to detect data-to-control plane saturation attack and limit the uploading rate of `packet_in` messages from the data plane cache.

To summarize, the contributions of our paper include the following:

- We deeply study the behaviors of the data-to-control plane saturation attack and analyze its impact on different OpenFlow controller applications.
- We design FLOODGUARD, a scalable, efficient, lightweight, and protocol-independent defense framework for SDN networks to prevent *data-to-control plane saturation attack* by using *proactive flow rule analyzer* and *packet migration*. Proactive flow rule analyzer module provides a new approach that combines symbolic execution and dynamic application tracking to derive proactive flow rules in runtime. Packet migration module migrates, caches, and processes table-miss packets by using rate limiting and round robin scheduling.
- We implement a prototype system and test it in different attack scenarios in both software and commodity hardware OpenFlow switch environments. We show the evaluation results of the protection of both the control plane and the data plane. Experiments show that FLOODGUARD provides a scalable and efficient security solution for SDN networks against data-to-control plane saturation attacks.

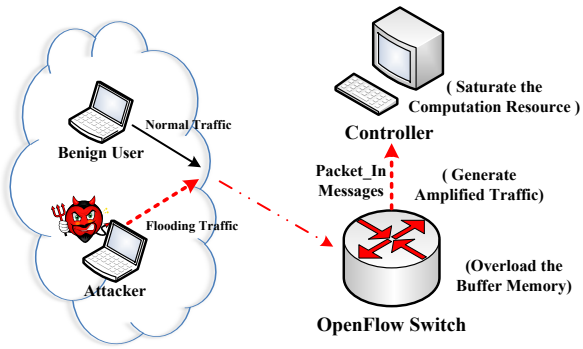

Fig. 1. Attack Process

The structure of this paper is as follows. We state the research problem and challenges in Section II. Section III introduces some related work. The design of FLOODGUARD is detailed in Section IV. The implementation and evaluation of FLOODGUARD are in Section V. At last we discuss and conclude our work in Section VI.

II. PROBLEM STATEMENT

In this section, we introduce some background knowledge about SDN, provide an adversary model of the data-to-control plane saturation attack, analyze the vulnerability in different OpenFlow applications and state our research problem.

A. Background on Flow Rule Installation

In OpenFlow networks, forwarding devices handle network flows based on the flow rules received from a controller. The controller installs flow rules to data plane in two approaches, i.e., proactively and reactively. In the *proactive* flow installation approach, the controller could populate the flow rules before all traffic comes to the switch. In the *reactive* flow installation approach, the controller could dynamically install or modify flow rules. The reactive approach enables the flexible management of forwarding behaviors based on current network situation. Thus, it could support more dynamic applications than the proactive approach. Currently most OpenFlow enabled networks choose the *reactive* approach for their management. In our work we focus on reactive controllers and consequent security threats against them.

B. Adversary Model

We assume an adversary could produce a large number of microflows to an OpenFlow-enabled network by her own host or controlling many distributed bot hosts. The attack traffic can be mixed with normal traffic and is hard to distinguish. The control plane and the data plane will suffer from the saturation at the same time and their resources will be consumed in a short time.

We start from a simple scenario to illustrate how an adversary can flood the SDN infrastructure. There is an OpenFlow-enabled switch which receives an external input stream. The stream is mixed with traffic from both a benign host and a bot host which is controlled by an attacker. The attacker could generate flooding traffic and consequently launch the saturation

Application	arp_hub	ip_balancer	route
Policy	LLDP packet → drop ARP packet → broadcast	srcip=1*, dstip=10.0.0.1 → dstip=192.168.0.1 srcip=0*, dstip=10.0.0.1 → dstip=192.168.0.2	dstip=192.168.0.1 → port(1) dstip=192.168.0.2 → port(2)

TABLE I. SAMPLE APPLICATIONS

Controller Platform	Packet_In Handler Function	Listening Interface
NOX	def packet_in_callback(self, dpid, inport, reason, len, bufid, packet)	core.register_for_packet_in
POX	def _handle_PacketIn (self, event)	core.openflow
Ryu	def _packet_in_handler(self, ev)	controller.ofp_event.EventOFPPacketIn
Beacon	public Command receive(IOFSwitch sw, OFMessage msg)	beaconcontroller.core.IOFMessageListener
Floodlight	public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx)	core.IOFMessageListener
OpenDayLight	public PacketResult receiveDataPacket(RawPacket inPkt)	sal.packet.IListenDataPacket

TABLE II. PACKET_IN HANDLER FUNCTIONS IN DIFFERENT CONTROLLERS

attack to the OpenFlow switch. Here is a basic and typical process of flow control in OpenFlow switches. When a table-miss occurs, which means there is a new packet which data plane does not know how to handle, the data plane will buffer the packet and send a `packet_in` message which contains the packet header to controller if the buffer memory is not full. If the buffer is full, the message will contain the whole packet instead of only its header. The attacker can exploit it by launching dedicated data-to-control plane saturation attack that floods SDN networks. She can generate a large number of anomalous packets, which means all or part of fields of each packet are spoofed as random values. These spoofed packets have a low probability to be matched by any existing flow entries in the switch. As a result, the flooding attack will significantly downgrade the performance of the whole OpenFlow infrastructure.

Figure 1 illustrates the basic process of the attack. The first component which is affected is the OpenFlow switch. The buffer memory will be consumed soon and the throughput is affected a lot including the packet forwarding throughput and the data-to-control plane communication channel throughput. We have tested the impact of the saturation attack on an OpenFlow switch. In the Mininet [22] environment, a software switch is dysfunctional by about 500 packets/second of table-miss UDP traffic. A hardware switch is a little more capable, but still vulnerable. This part of evaluation and discussion will be proposed in Section V. Besides the data plane switch, the bandwidth of data-to-control plane communication channel will also be occupied. In OpenFlow Specification 1.4 [2], if the buffer memory of a switch is full, the `packet_in` message will contain the whole body of each table-miss packet. That means the attacker can generate amplified traffic to occupy the data-to-control plane bandwidth. Suppose the data plane link and communication channel have the same capacity, amplification attack allows the attacker to consume less resources but cause more harm. At last, the control plane will suffer from the saturation attack, because the controller has to process each `packet_in` message and then respond to the data plane. The flooding traffic can easily overload the computation capability of the control plane in a short time.

C. Motivation

If we are able to pre-install all flow rules into the data plane and discard all other table-miss packets, the security problem is solved. However, it is unrealistic due to the dynamics of network policies. Each control application is composed of distinct packet-processing policies. Some policies are dynamic which means they may vary from different network situations. For example, in a cloud network the routing policies should be updated when the topology changes. Thus, the controller has to update the flow rules when network state changes. The dynamic nature makes it impossible to pre-install all flow rules. Therefore, the application needs to analyze data plane messages and update its packet-processing policies. As described above, the dynamic nature can result in both the scalability issue and security vulnerability.

For those packet-processing policies that will be dynamically changed inside an OpenFlow application, we define them as `dynamic` policies. Conversely, we define the unchanged policies as `static` policies. For example, we suppose there is a developer who deploys three applications, as shown in Table I, to manage a small network. The `arp_hub` application is to drop all Link Layer Discovery Protocol (LLDP) packets and broadcast Address Resolution Protocol (ARP) packets. The `ip_balancer` application is to load balance the traffic destined to a public IP address and split the traffic based on the source IP address. Incoming traffic with a source IP address whose highest-order bit is 1 gets a private destination IP address 192.168.0.1 and is forwarded to one server replica. The remaining traffic gets another private destination IP address 192.168.0.2 and goes to the other server replica. The third one, the `route` application generates routing path based on the destination IP address. Packet-processing policies in the `arp_hub` module are quite stable. However, those policies in the other two modules may update when topology changes. According to our definition, policies in the first module are static policies and those in the other two modules are dynamic policies. The reason why there exists dynamic policies is that there should be variables sensitive to the network state in the control application program. For example, the routing table in the third module is a state sensitive variable which is associated with the current network topology.

We argue that the dynamic policies make applications vulnerable, since dynamic policies need to be updated during the transition of the data plane. In current OpenFlow protocol, the controller obtains the transition information of the data plane mainly from the `packet_in` messages. Hence, during the flooding attack, a large number of `packet_in` messages will consume the resource in the controller and, simultaneously, even mislead the control plane. Also, the dynamic policies will change unpredictably and frequently, which makes it hard to predict the trend of them. We suppose at any time we are

able to know all the static policies and dynamic policies based on current network state, and then we can know what kinds of `packet_in` messages the control plane is able to process immediate responses to the data plane. We cannot simply drop other messages because many applications are learning-based and some messages that have not been learned by the applications may be useful in the future. These messages can be handled later when the controller becomes relatively idle after the attack. That kind of information will be quite useful for the defense. For further description about our defense solution, we introduce a new concept, i.e., proactive flow rules.

Proactive Flow Rules are all data-plane-level flow rules which can be generated by an application based on current controller state. In the above case in Table I, each policy could generate one entry of flow rule. At a certain moment, we call all these possible flow rules as proactive flow rules. The proactive flow rules are dynamic and time-sensitive. At another moment, the proactive flow rules may be different because the policies have changed. Proactive flow rules represent the range of `packet_in` messages which current control logic can handle at this moment. This concept is motivated from DIFANE [32] which introduces another similar concept called *low-level authority rules*. The authority rules are cached in the Authority Switches and need to be updated to handle dynamics. DIFANE caches these kinds of flow rules to keep packet processing in the data plane. We assume most of the flooding packets are out of the control logic and we aim to use the proactive flow rules to roughly separate the benign packets and malicious packets. Moreover, there are still several issues in DIFANE. For instance, there is no systematical solution to generate and update the proactive flow rules dynamically. Our work attempts to design a systematical solution and we will discuss it in Section IV.

We also conduct a deep analysis of the OpenFlow program model. Typically each control application contains a *packet_in handler function* to handle *packet_in* messages from the data plane. We summarize some popular controller platforms and their corresponding handler functions in Table II. The handler functions have a variety of names in different controller platforms, but have similar features. The handler function is event-driven. Triggered by `packet_in` messages, the function then may take some actions to handle this packet and consequent flows. Even for the same input, the handler function of an application could enforce different actions. As mentioned above, it is because of the dynamic nature of the state sensitive variables. For example, in the `route` application in Table I, the routing table is associated with the current network configuration and is state sensitive. Therefore, if we want to get the proactive flow rules, we need to know the current value of all state sensitive variables dynamically. We design a hybrid symbolic execution algorithm which is described in Section IV.

D. Research Challenges

The first one is to preserve major functionality of the network infrastructure when the saturation attack occurs. To achieve this, we design a new functional module called *proactive flow rule analyzer*, which is implemented in the control plane. It includes symbolic execution and dynamic application tracking to derive proactive flow rules in runtime. The proactive flow rule dispatcher component will install the proactive flow rules directly into the OpenFlow switch.

The second challenge is to handle table-miss packets without sacrificing benign packets. Simply dropping table-miss packets seems one answer but obviously not a good one since it could drop some benign packets. Therefore, we migrate the table-miss packets to a data plane cache when the attack occurs, and then trigger `packet_in` messages back to the controller in a limited rate.

Our design meets the following objectives. First, our framework is lightweight, i.e., under normal circumstances, only the monitoring component is active but others keep dormant. Second, our design is transparent to the controller applications and end hosts. Third, we merely add reasonably low overhead and latency. Finally, our solution is independent of the protocol of attack traffic (unlike AvantGuard which only defends against TCP-based flooding attacks).

III. Related Work

SDN Scalability: The data-to-control plane saturation attack is derived from the scalability problem in SDN research, which is how the control plane handles a large-scale network. Several papers have already studied this challenge. Onix [20] provides a distributed controller solution. Onix introduces a logically centralized but physically distributed controller framework which can share the work load. DIFANE [32] presents an approach to proactively compute low-level flow rules, distribute and then cache these rules into the data plane to handle a large number of incoming packets. However, it is not easy to directly apply this approach to our problem. We focus our problem domain on a reactive model while DIFANE uses a different model (it assumes proactive flow rules are given). Furthermore, a large number of generated fake packets will still cause high communication and computation burden in DIFANE. Our approach is a lightweight solution and provides packet-level migration, which guarantees the transparency to controller applications and end users. DevoFlow [14] introduces mechanisms for devolving control to a switch and finding elephant flows and micro-flows, which benefits to measurement requirement.

SDN Software Analysis: Some studies provide methods to analyze SDN control plane software. Pyretic [23] introduces some features to describe the model of an application from the abstraction point of view. Some existing studies such as NICE [11], VeriCon [7] and [16] propose several methods to verify different features of control applications. In [26], the authors improve the performance of control software troubleshooting by using a minimal causal sequence of triggering events.

SDN Security: SDN security becomes a hot research topic in recent years. There are two main directions. SDN-supported security research targets to use SDN technique (which is relatively a new technique) to solve traditional security challenges. Some papers such as Mahout [13] use traditional statistic based aggregation solutions to prevent flooding attacks in OpenFlow networks. These attacks are targeting the end hosts while the data-to-control plane saturation attack is against the OpenFlow network infrastructure. In addition, statistic aggregation algorithms do not help because the data-to-control plane saturation

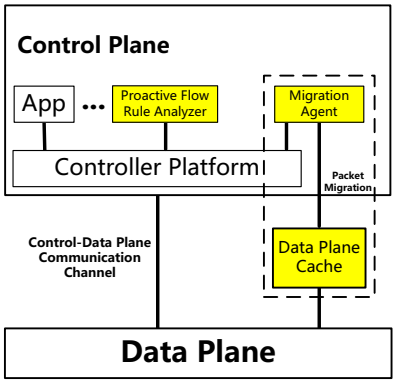

Fig. 2. Conceptual Architecture

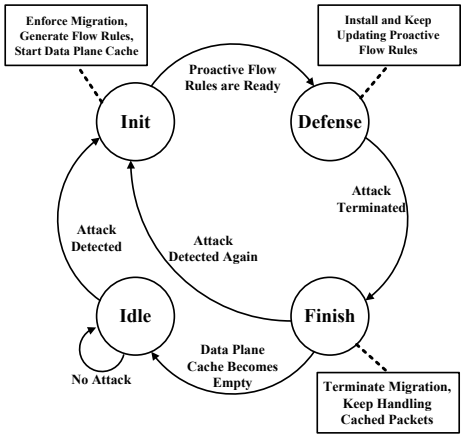

Fig. 3. States of FLOODGUARD

attack utilizes micro-flows. Therefore, we argue that statistic-based solutions are not suitable for our problem. CloudWatcher [30] introduces the Network Security Virtualization service to cloud networks. FRESCO [31] proposes a framework designed to facilitate the rapid design and modular composition of security applications. OpenSafe [8] improves the management of network monitoring applications.

Another direction is security for SDN which aims to protect and strengthen SDN-enabled infrastructure. Our work belongs to this direction. AvantGuard [29] attempts to solve the same saturation attack challenge, and it is the closest work to ours. AvantGuard introduces a module which implements a SYN proxy and only exposes those flows that finish the TCP handshake. AvantGuard can effectively defeat TCP based saturation attacks. Its limitation is obvious that it is invalid to other protocols. Our approach aims to defeat more generic saturation attacks in SDN, not only limited to TCP protocol. FortNOX [25] introduces the tunneling attack and proposes a security enforcement kernel to defend against this attack. Rosemary [28] introduces a sandbox-based framework to safeguard the SDN control layer against malicious or faulty control applications. TopoGuard [19] studies the network topology poisoning attack and proposes an extension to mitigate against the attack. Sphinx [15] proposes a framework to detect known and potential attacks on SDN networks.

IV. DESIGN

To address the security problems discussed in previous sections, we introduce FLOODGUARD, a scalable, efficient, lightweight and protocol-independent defense framework for SDN networks to prevent data-to-control plane saturation attack. We present the detailed design of FLOODGUARD in this section.

A. System Architecture

FLOODGUARD introduces two new functional modules to existing OpenFlow infrastructure : 1) a *proactive flow rule analyzer* module, and 2) a *packet migration* module. The analyzer module is to enforce the major functionality of the network infrastructure when the saturation attack occurs. The packet migration module is to transmit benign network flows to the OpenFlow controller without overloading it. A conceptual architecture of FLOODGUARD is shown in Figure 2. The *proactive flow rule analyzer* module is implemented as a controller application above the controller platform. In the *packet migration* module, the migration agent component is also implemented as a controller application and the data plane cache component sits between the control plane and data plane.

We maintain a finite-state machine to manage the whole FLOODGUARD system. The state machine is shown in Figure 3. Before all the states, FLOODGUARD has some preparation work, i.e., using symbolic execution to generate a set of path conditions for each `packet_in` handler function of each application. Compared with traditional symbolic execution method, we not only symbolize the input variables but also symbolize the global variables used in the `packet_in` handler function. After the preparation work, FLOODGUARD starts from the Idle State. Initially, if there is no attack, both the proactive flow rule analyzer and the packet migration modules keep idle. When a saturation attack is detected, FLOODGUARD comes to the Init State. The migration agent component starts to redirect the table-miss packets to the data plane cache. The proactive flow rule analyzer module will dynamically track the running controller applications and convert the path conditions which are generated before to proactive flow rules. At the same time, the data plane cache component begins to handle cached packets and generate `packet_in` messages to the controller. When the proactive flow rules are ready, FLOODGUARD comes to the Defense State. The analyzer module directly installs these rules to the data plane switches and keeps updating these flow rules. When the attack is detected to be over, FLOODGUARD comes to the Finish State. The migration agent component stops migrating the table-miss packets and the data plane cache will keep handling unprocessed packets. When the data plane cache finishes processing all cached packets, it will become idle again. Our framework does not need any modification to existing SDN infrastructure and is transparent to both the control plane and data plane.

B. Proactive Flow Rule Analyzer

The proactive flow rule analyzer module is running as an application in the controller and consists of three components: (i) symbolic execution engine, (ii) application tracker, and (iii) proactive flow rule dispatcher. The architecture of the proactive flow rule analyzer module is shown in Figure 4.

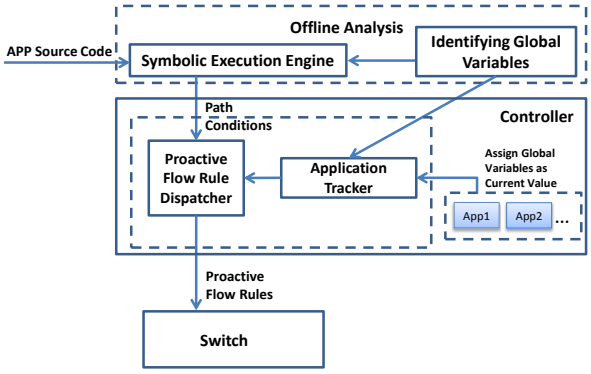

Fig. 4. Proactive Flow Rule Analyzer

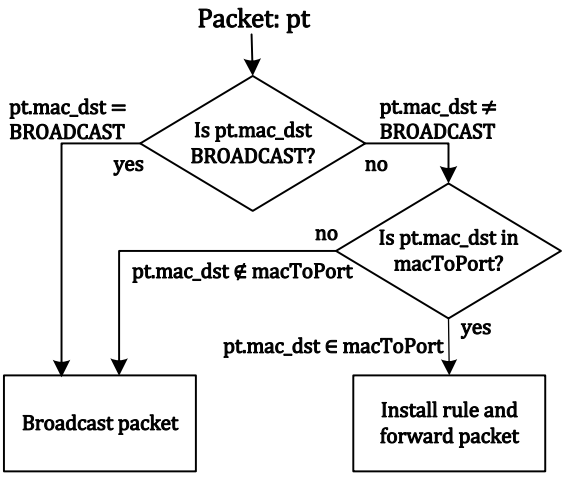

Fig. 5. Sample Control-Flow Logic

The proactive flow rule analyzer module can be activated anytime needed. For example, it can be activated right after the detection of the saturation attack, which is informed by (the flooding detection function in) the packet migration module. Once activated, the analyzer module generates the proactive flow rules and directly installs these rules into the data plane switches. Then the analyzer module keeps updating the proactive flow rules dynamically. In essence, the analyzer will leverage the logic of applications in the controller to generate proactive flow rules, which, to a great extent, covers all the possible upcoming packets that the application cares about. The challenge here is how to dynamically generate proactive flow rules. We introduce a new approach which combines symbolic execution and dynamical application tracking to address this challenge.

In a reactive controller, the controller platform maintains the connections to the data plane and transforms OpenFlow messages into events. Upon the controller, OpenFlow applications provide multiple event handlers to process OpenFlow messages (e.g., PortStatus, PacketIn, Barrier). In this paper, we only focus on the `packet_in` event handler. It is because from Section II, we know that the `packet_in` handler function is the main target of flooding attacks. The input of `packet_in` handler function is the `packet_in` event and the output is flow rules. We use a sample application, l2_learning application [5], and describe the corresponding control flow graph of its `packet_in` handler. First, we briefly describe the logic of this function. The input for this function is the `packet_in` event. This function maintains a MAC-port mapping table, which can be learned from the source MAC address and incoming port of previous `packet_in` messages. The function firstly checks if the destination MAC address of the packet is a broadcast address. If so, the function just simply broadcasts the packet. If not, the function will search this MAC address in the MAC-port mapping table. If this MAC address has not been learned before, the function has no idea which port to forward it so just broadcast it. If the MAC address has been learned before, the function installs a relative flow rule and forward this packet to the mapping port.

The control-flow logic of `packet_in` handler function of the l2_learning is as follows. One for input whose destination MAC address is broadcast ($pt.mac\_dst = BROADCAST$), one for input whose destination MAC address is not broadcast and not learned before ($pt.mac\_dst \neq BROADCAST$ and $pt.mac\_dst \notin macToPort$) and the other for input whose destination MAC address is not broadcast but has been learned before ($pt.mac\_dst \neq BROADCAST$ and $pt.mac\_dst \in macToPort$). The $BROADCAST$ is a constant value, and the value of the data structure $macToPort$ is network state sensitive [2]. From our analysis and discussion in Section II, we identify the variable $macToPort$ as a *state sensitive variable*. Suppose at a certain point in the running cycle, $macToPort$ has a concrete value (i.e., $\{0x00000000000A : 01\}$. Consequently in the control-flow logic the first two branches only generate `packet_out` messages without any flow rules. Nevertheless, the third branch may generate a flow rule: $mac\_dst = 00:00:00:00:00:0A$, $action = output : 01$, which is the proactive flow rule we can get at this moment.

Symbolic Execution [9], [18] is a practical way to generate proactive flow rules. Symbolic Execution is a program analysis approach, which is capable of efficiently traversing possible branches in a program. A symbolic execution engine will symbolize the input of a program and then execute all the feasible paths at the beginning of the program. During executing each path, the engine records the accumulation of conditions that lead to this path, which is called "path conditions" or "path constraints". For example, in l2_learning application, the path condition for the third branch is $pt.mac\_dst \neq BROADCAST$ and $pt.mac\_dst \in macToPort$. When the engine finishes the execution of all feasible paths, we get all path conditions of the program.

For the sake of reducing runtime overhead, we choose to run symbolic execution offline for generating proactive flow rules. However, simply offline running symbolic execution on the `packet_in` handler function as mentioned above cannot totally solve the problem. It is because `packet_in` handler function may contain state sensitive variables whose value will dynamically change. For example, in the l2_ learning application example, the initiate value of $macToPort$ is empty. We can only assign $macToPort$ as its initial value,

[2]The key reason it is network state sensitive is because it will dynamically change and vary in every call back of the handler function.

which means we will lose the third branch in the generated path conditions. To tackle the problem, we increment symbolic execution with dynamic application tracking. In detail, when we generate the path conditions offline, we symbolize both the input variables and the state sensitive variables. Then we use dynamic application tracking to locate state sensitive variables and extract them at runtime to derive proactive flow rules. In the l2_learning example, we first symbolize the input variable (i.e., `packet_in` event) and the $macToPort$. We get path conditions which are the three branches. When the saturation attack happens, we keep tracking the application to get the runtime value of $macToPort$ and assign to the path conditions. Then we dynamically convert the path conditions to proactive flow rules.

To derive proactive flow rules, we need to locate state sensitive variable, which is highly related to program dynamics. For example, $macToPort$ is a state sensitive variable. We find that all state sensitive variables are global variables to the function, which means to the handler function, the set of global variables is a superset of the state sensitive variables. The application program will call the handler function hundreds of times with different value of these variables. Therefore, we decide to symbolize input variables and all global variables used in the handler function to generate the path conditions. Then we can assign the value of these global variables dynamically to the path conditions and then generate our needed proactive flow rules. Motivated by this idea, we introduce a new approach which combines the symbolic execution and dynamic application tracking to meet our requirement. We summarize the algorithm of our approach as follows.

Algorithm 1 Generation of the Path Conditions (offline)

Input: F: packet_in handler function
Output: P: a set of path conditions
1: $input \leftarrow \emptyset$, $global \leftarrow \emptyset$
2: $input$ = find_input_variables(F)
3: $global$ = find_global_variables(F)
4: F' = symbolize(F, $input$, $global$)
5: P = symbolic_execution_engine(F')
6: **return** P

Algorithm 2 Converting to Proactive Flow Rules (runtime)

Input: P: a set of path conditions, $global'$: global variables with real-time value assigned
Output: R: proactive flow rules
1: $R \leftarrow \emptyset$
2: $paths$ = assign_value(P, $global'$)
3: **for** each path condition $p \in paths$ **do**
4: **if** $p.decision$ = Modify_State_Message **then**
5: $R.add$ (convert($p.path\_condition$))
6: **end if**
7: **end for**
8: **return** R

We first separate the whole process into two steps. In the first step, our input is the `packet_in` handler function and our desired output is the path conditions of this function. We first find input variables (e.g. `packet_in` event) and the global variables which are used in the `packet_in` function. Next we symbolize both the input variables and the global variables. Then we use traditional symbolic execution algorithm to traverse possible branches, collect all path conditions and then generate the path conditions. This step will be relatively time consuming. However, the symbolic execution engine component could process this step offline in advance, which means it will not increase overhead to our system. This step is summarized in Algorithm 1.

The second step is dynamic analysis. In the state machine, when it goes to Init State, the application tracker component will process the second step. It will track and assign current value of the global variables to the path conditions. After this process, in the path conditions only input variables are symbolized. Then we use the proactive flow rule dispatcher component to analyze each path condition. We only consider the paths whose final handling decision is in a small set which is to generate Modify State Message (defined in OpenFlow Spec. 1.4.0) [3]. At last we get the proactive flow rules that we want. This step is summarized in Algorithm 2. Figure 4 illustrates the process of dynamically generating proactive flow rules. In the l2_learning case as shown in Figure 5, the number of proactive flow rules is based on how many MAC-port pairs have been learned in the $macToPort$. After the proactive flow rules are ready, the analyzer component will install them to the data plane switches.

C. Packet Migration

Installing proactive flow rules during the attack will preserve the major functionality of the network infrastructure because those packets that mach these flow rules are what the SDN apps mainly care about. For the unmatched packets (i.e., table-miss packets), one may think that we could simply drop them. However, in that case, some new network flows will not be monitored by the controller and thus be dropped. For example, in the above l2_learning application example, when we generate the proactive flow rules and install them into the switches, we may sacrifice the learning capability. After installing the proactive flow rules, new incoming network flows cannot not be learned by the controller. That is because the new incoming flows have not been learned before and thus cannot match any proactive flow rules. Therefore, we cannot simply drop the table-miss packets. That leads to our second challenge, i.e., how to handle the flooding packets without sacrificing benign packets? Our idea is to temporarily cache the table-miss packets after the installation of the proactive flow rules. We introduce the *packet migration* module. The packet migration modules contains two components: migration agent and data plane cache .

*1) **Migration Agent**:* The migration agent component is the "brain" of the whole FLOODGUARD system. It has three main functions. The first function is to detect the saturation attack. Anomaly-based flooding detection is easy to get around by an attacker who is willing to slowly execute the attack. Only using real-time rate of `packet_in` messages is not enough to detect the attack. Therefore, our detection algorithm makes use of both the real-time rate of `packet_in` messages from the data plane and the utilization of the infrastructure (buffer memory, controller memory and CPU) to calculate current usage percentage of the capacity of our OpenFlow

[3]This kind of messages will install new flow rules in data plane switches.

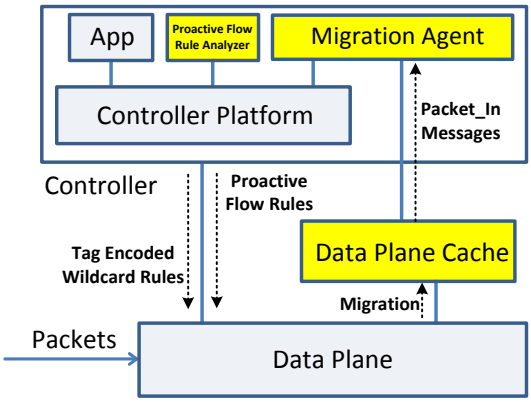

Fig. 6. Migration Process

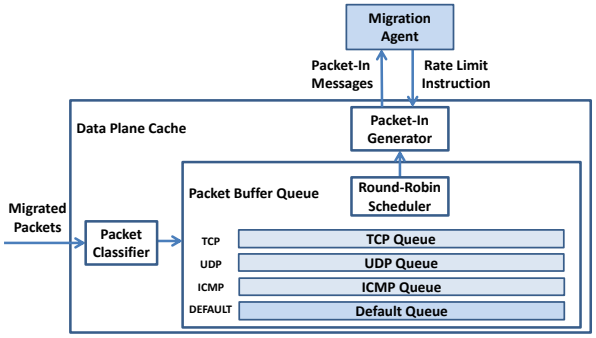

Fig. 7. Data Plane Cache

network. We identify there is a potential flooding attack based on certain anomaly threshold. When the migration agent detects the saturation attack occurs or ends, it will trigger the corresponding state transition in the state machine described above.

The second task of the migration agent component is to migrate table-miss packets to the data plane cache. When the saturation attack is detected, it will change the system state to the Init State, which will trigger both the proactive flow rule analyzer and the data plane cache. The migration agent component installs one entry of wildcard flow rule which has the lowest priority and forwards all table-miss packets to data plane cache. Therefore, the flooding packets will not overload the switch or flood the controller. The data plane cache will temporarily store all the table-miss packets. Some packets will be given priority to trigger `packet_in` messages from the data plane cache.

However, the process has one obvious challenge. In OpenFlow protocol, a `packet_in` message contains both packet header and INPORT information. Due to the addition of data plane cache, the original INPORT information is lost in the process of migration. When we generate `packet_in` messages later in the data plane cache, we lose the original INPORT information. To solve this problem, we utilize a packet tag to preserve INPORT information. In OpenFlow specification, some fields are used for matching packet headers. We can borrow some reserved fields to tag table-miss packets (e.g. 8 bits TOS). Therefore, we modify the wildcard flow rule mentioned above which is to migrate the table-miss packets. In our implementation, we install multiple wildcard rules instead of only one rule. When the controller detects the flooding attack, it installs several wildcard rules to migrate table-miss packets with encoded INPORT information into tag. For each port there is a corresponding wildcard rule. We add one matching condition which matches the incoming port and also add one action which preserves the INPORT information to the TOS field. For example, one wildcard rule could be: "inport = 1, actions : set-tos-bits = 1, output : data_plane_cache". If the ingress switch has 6 ingress ports, we need 3 bits to encode INPORT information and the number of wildcard rules is 6. In the data plane cache, when we need to generate `packet_in` messages, we can decode the INPORT information from the TOS field. The whole process is shown in Figure 6.

When the data plane cache generates `packet_in` messages, it cannot directly send the messages to the controller. It is because the controller platform distinguishes different datapaths (switches) from different connections (e.g., TCP session). If the data plane cache directly sends messages to the controller, the data plane cache will be identified as a new datapath. The third function of the migration agent component is to solve this issue. The data plane cache will send the `packet_in` messages to the migration agent component. Then the migration agent will raise `packet_in` events with the original datapath information to other applications in the controller. Also, the migration agent instructs rate limit to data plane cache based on the utilization of system resource and the rate of incoming `packet_in` messages.

*2) **Data Plane Cache**:* Data plane cache is a machine/device that temporarily caches table-miss packets during the saturation attack. During the data-to-control saturation attacks, most of the flooding traffic will be redirected the data plane cache instead of flooding the OpenFlow infrastructure. The data plane cache, as shown in Figure 7, has three functions: packet classifier, packet buffer queue, and packet_in generator. When a migrated table-miss packet arrives in data plane cache, the packet classifier parses the header of the packet and attaches it to its corresponding buffer queue. Specifically, there are four packet buffer queues inside data plane cache based on the packet protocol, i.e., TCP, UDP, ICMP, and Default. Each packet buffer queue is based on FIFO (first-in, first-out) and embraces tail drop scheme, i.e., when new packets come to a full packet buffer queue, the earliest coming packet inside the packet buffer queue will be dropped. Among those four packet buffer queues, we adopt round-robin scheduling algorithm to pick the queue header packet for future generating `packet_in` messages. The insight behind the round-robin-based packet buffer queue lies in the observation that an attacker normally exploit a specific protocol to launch attacks, e.g., TCP SYN flooding attack, UDP flooding attack, and ICMP flooding attack. Even if the attacker knows how our scheduling manner works and attacks the various protocols, the effect of the manner is the same as just using one queue and has no drawbacks. Lastly, the packet_in generator converts the scheduled packets into `packet_in` messages by decoding the INPORT information from the tag added before and sends them to the OpenFlow controller. The sending rate of packet_in generator is controlled by the migration agent inside the remote OpenFlow controller.

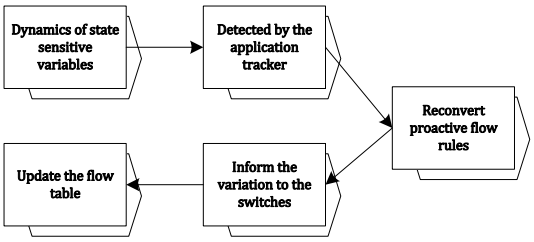

Fig. 8. Handling Dynamics

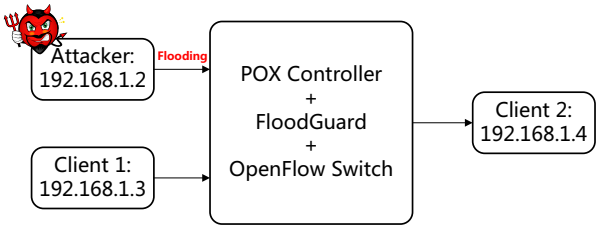

Fig. 9. Test Topology

D. Handling Dynamics

During the flooding attack, proactive flow rules may vary due to network dynamics. For example, in the ip_balancer application, the change of traffic in different flows will lead to the re-generating of flow rules. If we still use previous proactive flow rules in the data plane, it will lead to unpredictable results. Therefore, the proactive flow rules should keep consistent with current network state. To handle the dynamics, in the proactive flow rule analyzer module we constantly update the proactive flow rules. We have several steps to update the proactive flow rules, as illustrated in Figure 8. By frequently referencing to the global variables, any change of value can be easily identified by the application tracker component. Then this component can find corresponding path conditions and regenerate the proactive flow rules. At last, the control plane will update the latest proactive flow rules to the data plane switches. The variation should be quite simple as adding or removing a few matching rules. In the case shown in Table I, if the incoming traffic with a source IP address whose highest-order bit is 1 gets a private destination IP address which changes to 192.168.0.2 and the remaining traffic changes to 192.168.0.1, the change can be detected by the analyzer. Then the proactive flow rule analyzer regenerates the proactive flow rules and reports the variation of two flow rules to the data plane switches. Consequently, the flow table in the OpenFlow switch has the latest matching rules which represents the current network state.

There is a tradeoff between the performance and the accuracy. We can update the rules every time after a change. That will bring high accuracy, but also introduce relatively high overhead. We can also update the rules after a certain number of changes happen. That will increase the performance but reduce the accuracy. We can also update the rules based on a constant time interval (e.g., 1ms). We think the decision should be based on the actual situation and system features.

E. Discussion

One issue in the deployment is that how many data plane caches are necessary for a large number of OpenFlow switches. Ideally, we only need to deploy one data plane cache to serve all switches. However, to be more scalable, we could also use a set of data plane caches, with each in charge of a subset of switches (e.g., a subnet of an enterprise network or a rack of a cloud network).

Another concern is that the size of TCAM memory in switches may not be enough to install all proactive flow rules. Note that in our design, we do not add new extra rules. Instead, we just proactively install those rules in advance. If the TCAM memory is really limited, we have another design option which is to install and update the proactive flow rules into the data plane cache. In the data plane cache, for those packets who can find a match, we provide them a higher priority to trigger `packet_in` messages. However, the system needs to sacrifice some performance for this design option. There is a tradeoff. According to the actual situation, the network administrator could make a decision between the two options.

V. Evaluation

In this section we introduce our implementation of FloodGuard and evaluate the performance and overhead of our framework.

A. Implementation

We implement FloodGuard and test into both software and hardware OpenFlow environments. In the software environment, we use Mininet [22] to emulate the OpenFlow-enabled network data plane. In the hardware environment, we use an OpenFlow-enabled commercial LinkSys WRT54GL switch with Pantou [4]. The proactive flow rule analyzer module is implemented as an application upon the POX controller [5], a lightweight OpenFlow controller platform. We modify the concolic execution engine in the NICE [11] project to implement our symbolic execution engine and use STP [17] as the path constraint solver. We implement the data plane cache as a software system in approximately 1,000 lines of C++ code.

B. FloodGuard Defense Effects

We first evaluate and compare two scenarios: (i) testing an application with existing OpenFlow network, and (ii) testing the same application with FloodGuard. We have two kinds of test environments. One is a hardware switch environment that includes a POX controller, an OpenFlow switch (LinkSys WRT54GL), a server machine that implements data plane cache and three clients. We also have a software environment, in which we use Mininet [22], a popular SDN emulation tool. One client is the attacker who launches a UDP flooding attack to the switch. The other two clients keep normal communicating with each other. The POX controller is running the l2_learning application which discovers the topology and provides basic forwarding services. The test environment is shown in Figure 9.

We first measure the bandwidth between the two clients with and without flooding attacks. The attacker keeps generating different rates of UDP flooding traffic to the switch. We evaluate the impact on bandwidth under different attack rates with and without using FloodGuard. Since software

Application	State Sensitive Variable (Type)	Description
l2_learning	macToPort (dictionary)	{mac address : INPORT}
l3_learning	arpTable (dictionary)	{IP address : mac address and INPORT}
ip_balancer	servers (list)	IP addresses of duplicated servers
	service_ip (IPaddr)	IP addresses of services
	live_servers (dictionary)	{IP address : mac address and INPORT} of live duplicated servers
	memory (dictionary)	{four tuples of packet header : flow record}
of_firewall	firewall (dictionary)	{packet patterns : TRUE or FALSE}
	table (dictionary)	{switch data path : mac address}
mac_blocker	blocked (set)	mac addresses of blocked hosts

TABLE III. THE STATE SENSITIVE VARIABLES IN APPLICATIONS

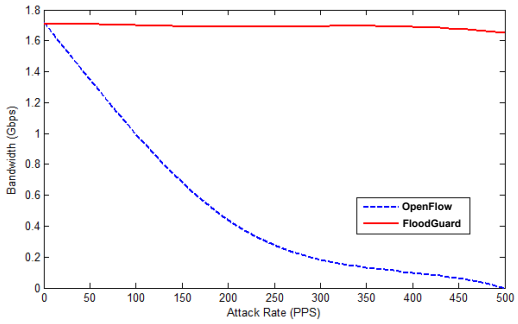

Fig. 10. Bandwidth in Software Environment

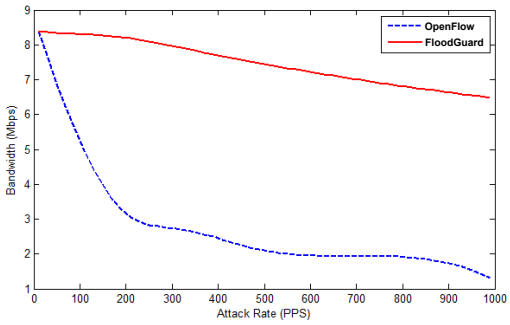

Fig. 11. Bandwidth in Hardware Environment

switches and hardware switches have different performance and capacity, we measure the bandwidth in both software and physical switch environments. We use an open source tool *iperf* to measure the available bandwidth between two clients.

The results in software environment and hardware environments are shown in Figure 10 and Figure 11. In our design, without the saturation attack, FLOODGUARD should have no impact on the data plane. This has been verified in our bandwidth evaluation. FLOODGUARD does not affect the bandwidth of traffic forwarding. In the software environment, without FLOODGUARD, the bandwidth is about 1.7Gbps when there is no attack. When we start the saturation attack and increase the attack rate, the bandwidth goes down quickly. The bandwidth decreases in half after about 130 packets per second (PPS) of traffic. The whole network is dysfunctional after an attack rate of 500 PPS. While using FLOODGUARD the bandwidth also starts from about 1.7Gbps without the attack. Even though we increase the attack rate up to 500PPS (Packets Per Second), the bandwidth keeps almost unchanged. In this sense, FLOODGUARD can protect the software switch well.

In the hardware environment, the results also show the good protection of FLOODGUARD. Without FLOODGUARD, the bandwidth also quickly goes down with the increase of the attack rate. The bandwidth starts from about 8.4Mbps and decreases in half after about 150 PPS. With an attack rate after 1000 PPS, the hardware switch is almost dysfunctional. By using FLOODGUARD, the bandwidth keeps as about 8.3Mbps under an attack of less than 200 PPS. While after a rate of 200 PPS, the bandwidth will also decrease slowly. That is because our switch does not have the ternary content addressable memory (TCAM) but instead uses the OpenWRT [3] firmware which implements a software flow table. The software flow table cannot reach the same level efficiency as TCAM. Even then, we can still see that FLOODGUARD provides significant good protection and saves more resources. It is expected that with a real OpenFlow hardware switch, FLOODGUARD will have better protection results.

Next we will show our evaluation of the protection impact on the control plane. We illustrate how FLOODGUARD can protect the controller. We choose five applications for our evaluation: l2_learning, ip_balancer, l3_learing, of_firewall and mac_blocker (we downloaded the first four applications from [5] and of_firewall from [1])). We simultaneously run these five application in the controller and use an attacker to launch the saturation attack with a rate of 100 PPS in the hardware switch environment. We keep monitoring the resource consumption of each application (we choose the CPU utilization of each application as the indicator of how many resources it consumes). In Figure 12 we show the evaluation results.

We can see the protection effects of FLOODGUARD in the figure. The flooding attack starts at about 0.6s. We can observe that the CPU utilization of each application increases quickly and reaches a peak at about 0.8s. Then the CPU utilization begins to go down slowly because of the installation of the migration flow rules. When we can also observe the impact of the data plane cache, the utilization does not go down immediately to the initial level. Instead, the utilization maintains at a medium level for some time. At about 1.5s, the CPU utilization of all the applications goes back to the the initial level. We can observe from the results that FLOODGUARD provides effective protection to the control plane. The saturation attack does not consumes many resources of the control plane.

C. Overhead Analysis

In this section we show our evaluation about the overhead of FLOODGUARD. First we measure the overhead of symbolic-execution engine and proactive flow rule dispatcher, which

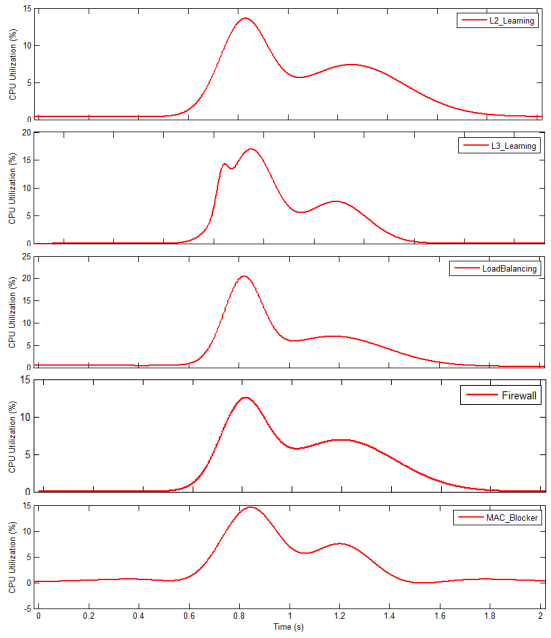

Fig. 12. CPU Utilization under the Flooding Attack

OpenFlow	OpenFlow + FLOODGUARD		
Total	Total	Data Plane Cache	After Migration
130ms	157ms	30ms	127ms

TABLE IV. AVERAGE DELAY OF THE FIRST PACKET IN EACH NEW FLOW

seem to be time-consuming components. To generate the path conditions, running symbolic execution engine for each application takes relatively long time which is more than ten seconds. However, from Section IV we know Algorithm 1 can be processed offline in advance, which means it will not add any overhead to the runtime performance of FLOODGUARD. Thus, the overhead of symbolic execution engine is not a concern.

At runtime, we need to dynamically dispatch proactive flow rules. This part of overhead cannot be omitted. We keep using the five controller applications. In Table III we provide the state sensitive variables in each application and their descriptions. For each application mentioned above, we test the average overhead of generating proactive flow rules. The results are shown in Figure 13. The logic in every application has different complexity. For most cases the overhead is less than 2ms. We can see that the worst case is about 9ms in of_firewall application. That is because this application contains relatively more complex data structure, which the proactive flow rule dispatcher takes more time to analyze the path conditions. The overhead is still acceptable for our system.

We also evaluate the average time delay of the whole migration process in the physical environment. We generate a new benign TCP connection under the UDP flooding attack and we will let the first handshake packet trigger table-miss (by not installing relevant proactive flow rules which may be generated at this time) in order to measure the migration overhead. The scenario is the same as shown in Figure 9 and

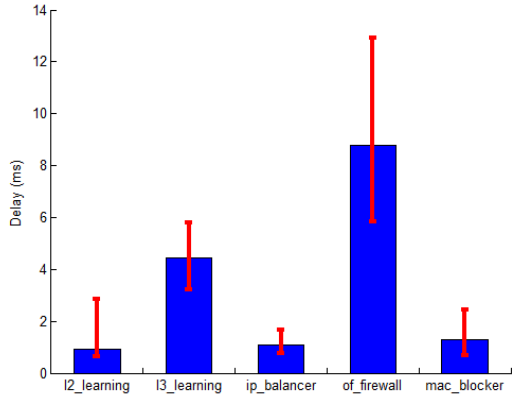

Fig. 13. Overhead of Generating Proactive Flow Rules

the result is shown in Table IV. Without flooding traffic it takes an average time about 130ms to process and forward the first packet of a new flow in the original OpenFlow network. However, when flooding attacks occur, the delay will become infinite. FLOODGUARD can detect and prevent such attacks meanwhile will inevitably bring some overhead. During the flooding attack, the first handshake packet of a new TCP connection will trigger table-miss and will be forwarded to the data plane cache with almost no delay. Since the system is under the UDP flooding attack, the TCP buffer queue in the data plane cache is relative idle. It takes about 30ms to process the packet. After that, the procedure of re-triggering a `packet_in` message, setting up new flow rules and forwarding the packet takes about 127ms. To sum up, FLOODGUARD increases about a total of 27ms overhead (20.8%) in this scenario. When new flow rules are set up in the switch, there is no more delay for the subsequent packets in this flow. Although FLOODGUARD unavoidably adds some overhead of time delay, the tradeoff is that it can cache flooding packets and protect both the controller and the switches. Thus FLOODGUARD makes the OpenFlow network more secure against flooding attacks.

VI. CONCLUSION

In this paper, we propose FLOODGUARD to prevent the data-to-control plane saturation attack by using *proactive flow rule analyzer* and *packet migration*. When the saturation attack is detected by FLOODGUARD, the packet migration module will redirect the table-miss packets in the OpenFlow switch to the data plane cache. At the same time, the proactive flow rule analyzer module will dynamically track the runtime value of the state sensitive variables from the running applications, convert generated path conditions to the proactive flow rules dynamically and install these flow rules into the OpenFlow switches. Then the data plane cache will slowly send the table-miss packets as `packet_in` messages to the controller by using rate limiting and round-robin scheduling algorithm. We present a prototype implementation tested in both a software environment and a commodity hardware OpenFlow switch environment with real attack scenarios. The evaluation results demonstrate the effectiveness of FLOODGUARD and show that our system only add minor overhead.

VII. Acknowledgements

This material is based upon work supported in part by the Air Force Office of Scientific Research (AFOSR) under Grant No. FA-9550-13-1-0077, the National Science Foundation (NSF) under Grant No. CNS-0954096 and a Google Faculty Research award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFOSR, NSF and Google.

References

[1] OpenFlow Firewall Application. https://github.com/hip2b2/poxstuff/blob/master/of_firewall.py.

[2] OpenFlow Specification v1.4.0. http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[3] OpenWrt: a Linux Distribution for Embedded Devices. https://openwrt.org/.

[4] Pantou: OpenFlow 1.0 for OpenWRT. http://www.openflow.org/wp/openwrt/.

[5] POX controller. http://openflow.stanford.edu/display/ONL/POX+Wiki.

[6] Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[7] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-Defined Networks. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[8] Jeffrey R. Ballard, Ian Rae, and Aditya Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. In *Proceedings of USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking*, 2010.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A System for Scalable OpenFlow Control. http://www.cs.rice.edu/ eugeneng/papers/TR10-11.pdf.

[11] M. Canini, D. Venzano, P. Peresini, D. Kostic, and Jennifer Rexford. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[12] Martin Casado. The Scaling Implications of SDN. http://networkheresy.com/2011/06/08/the-scaling-implications-of-sdn/.

[13] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout:Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *Proceedings of IEEE INFOCOM 2011 Conference*, 2011.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proceedings of ACM SIGCOMM 2011 Conference*, August 2011.

[15] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[16] Mihai Dobrescu and Katerina Argyraki. Software Dataplane Verification. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[17] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the International Conference in Computer Aided Verification (CAV 2007)*, July 2007.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[19] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[20] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. In *Proceedings of ACM SIGCOMM Computer Communication Review*, April 2008.

[22] Mininet. Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/.

[23] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[24] OpenFlow. Innovate Your Network. http://www.openflow.org.

[25] Phillip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A Security Enforcement Kernel for OpenFlow Networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, August 2012.

[26] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H.B. Acharya, K. Zarifis, and S. Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proceedings of ACM SIGCOMM 2014 Conference*, 2014.

[27] S. Shin and G. Gu. Attacking Software-Defined Networks: A First Feasibility Study (short paper). In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013.

[28] 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 *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS)*, 2014.

[29] 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 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[30] Seungwon Shin and Guofei Gu. CloudWatcher: Network Security Monitoring Using OpenFlow in Dynamic Cloud Networks (or: How to Provide Security Monitoring as a Service in Clouds?). In *Proceedings of the 7th Workshop on Secure Network Protocols (NPSec12), co-located with IEEE ICNP12*, October 2012.

[31] Seungwon Shin, Phil Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[32] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *Proceedings of ACM SIGCOMM 2010 Conference*, August 2010.