Packet Injection Attack and Its Defense in Software-Defined Networks

Shuhua Deng, Xing Gao, Zebin Lu, and Xieping Gao

***Abstract*— Software-defined networks (SDNs) are novel networking architectures that decouple the network control and forwarding functions from the data plane. Unlike traditional networking, the control logic of SDNs is implemented in a logically centralized controller which provides a global network view and open programming interface to the applications. While SDNs have become a hot topic among both academia and industry in recent years, little attention has been paid on the security aspect. In this paper, we introduce a novel attack, namely, packet injection attack, in SDNs. By maliciously injecting manipulated packets into SDNs, attackers can affect the services and networking applications in the control plane, and largely consume the resources in the data plane. The consequences could be the disruption of applications built on the top of the topology manager service and rest API, as well as a huge consumption of network resources, such as the bandwidth of the OpenFlow channel. To defend against the packet injection attack, we present PacketChecker, a lightweight extension module on SDN controllers to effectively detect and mitigate the flooding of falsified packets. We implement a prototype of PacketChecker in floodlight controller and conduct experiments to evaluate the efficiency of the defense mechanism. The evaluation shows that the PacketChecker module can effectively mitigate the attack with a minor overhead to the SDN controller.**

***Index Terms*— Software defined networks, packet injection attack, security.**

I. Introduction

SOFTWARE defined networks (SDNs) are new network architectures decoupling control logics and forwarding functions from network infrastructures. In SDNs, a logically centralized controller is employed to provide a holistic network view as well as network programmable interfaces. The controller collects the devices information (e.g., host location, switches information) and constructs a global topology view. Based on the topology view, the controller then deploys forwarding policies by installing flow rules on each networking devices (e.g., switches). Through the northbound application programming interfaces, networking applications built on the top of the controller can further customize network control policies such as firewalls, routing algorithms, and load balancing strategies. Such centralized controls significantly simplify the modification of network policies and the deployment of networking applications. OpenFlow [1], as one of the most successful SDNs protocols, has been widely deployed in realistic cloud computing environments to improve the reliability and performance of cloud services. Google reports that they achieved near 100% link utilization in their data centers using SDN principles and OpenFlow [2].

While SDNs bring huge successes, the idea of centralized control also raises new security concerns. As the core of the whole network, the SDN controller is responsible for the schedule of the entire network traffics. Therefore, a compromised controller would result in devastating consequences. Once the OpenFlow controller is under malicious attacks, the normal networking functions would be disrupted. Even worse, attackers who hijack the controller could control the entire network and easily alter every single flow. Moreover, since all applications rely on the global network view to function properly, the poisoned network view caused by malicious packets would lead to flow rule conflicts and policy violations. As a result, numerous efforts from both academia and industry have been made to enforce the security of SDNs. For example, FortNox [3], FLOWGUARD [4] and Fleet [5] are proposed to solve the problem of policy violations. AVANT-GUARD [6] and FloodGuard [7] are built to defend against the denial-of-service attacks on SDNs. While those works provide solid solutions which effectively raise the bar for launching various malicious activities such as the saturation attacks [6]–[8], they ignored malicious users who can intentionally inject packets to overwhelm the secure channels between the controller and forwarding devices, poison the global topology view and tremendously consume controller's resources.

In this paper, we unveil a new vulnerability exists in the OpenFlow protocol. In existing design of the communication processes, when a forwarding device (e.g., a switch) meets a new packet with no matching flow rules in the flow table, it sends a `Packet-In` message to the controller through the secure channel without any regulation mechanisms. We demonstrate that such a design could be abused by an adversary to launch packet injection attacks. A malicious user can continuously inject manipulated packets with random falsifications of parts of the header fields (e.g., MAC address), forcing the switches to send `Packet-In` messages to the controller. Those fake devices information would deceive the topology management service to generate a ghostly topology with huge amounts of non-existing devices included, and thus mislead the networking applications via the Rest API.

Manuscript received March 26, 2017; revised September 5, 2017; accepted October 12, 2017. Date of publication October 23, 2017; date of current version December 19, 2017. This work was supported in part by NSFC under Grant 61172171 and in part by the Hunan Provincial Postgraduate Research and Innovation Project of China under Grant CX2015B207. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Mauro Barni. *(Corresponding author: Xieping Gao.)*

S. Deng, Z. Lu, and X. Gao are with the Key Laboratory of Intelligent Computing and Information Processing of Ministry of Education, Xiangtan University, Xiangtan 411105, China (e-mail: shuhuadeng@163.com; xpgao@xtu.edu.cn).

X. Gao is with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23185 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2017.2765506

In certain cases, the attack can disable the functionality of applications and even crash the applications.

To defend against such attacks, we propose *PacketChecker* to prohibit malicious hosts from manipulating a large amount of `Packet-In` intentionally. *PacketChecker* acts as a lightweight extension to existing OpenFlow controllers. It filters out `Packet-In` messages according to the header fields of the packets before these messages processed by the controller core module. By comparing the `MAC address`, `Switch DPID` and `Inport` information carried in the `Packet-In` message, *PacketChecker* can automatically identify the injected packets. Once an anomaly is detected, it informs the switch to discard these packets through a `Flow-Mod` message, thus avoids the controller resources be exhausted. We implement a prototype of *PacketChecker* module in Floodlight controller.

To evaluate the impact of the packet injection attack on SDN controllers, as well as the effectiveness of *PacketChecker*, we conduct a set of experiments in the Mininet environments. We launch the packet injection attack on a simulated SDN network under three different networking topologies and compare the results with *PacketChecker* installed. Our results show that this attack can disrupt the networking applications and clog up network traffic. However, with *PacketChecker* integrated, the controller can effectively detect and drop all injected packets from malicious user on the switch. Also, our evaluation results show that *PacketChecker* induces minor performance overhead.

In general, our paper makes the following contributions.

- We uncover a new security vulnerability in the OpenFlow communication process, topology management service and Rest API in the SDN controller.
- We present a novel attack, namely the packet injection attack, against the SDN controller. We demonstrate the feasibility of such attack on the Mininet [9] emulation environment and analyze the consequences.
- We discuss several possible solutions to mitigate the new threat. Based on some features of SDNs, we design and implement *PacketChecker* by extending the Floodlight controller to prevent the packet injection attack.
- We conduct experiments to evaluate *PacketChecker* in terms of the effectiveness and efficiency. The experimental results show that *PacketChecker* can effectively mitigate the packet injection attack while imposing negligible overheads.

The rest of the paper is organized as follows. We present the background information about SDNs and the OpenFlow protocol in Section II. We describe the vulnerability in SDN controller and introduce the packet injection attack in Section III. We discuss the countermeasures against the packet injection attack and propose *PacketChecker* in Section IV. We detail the implementation and evaluation of *PacketChecker* in Section V. We survey the related work in Section VI. Finally, we conclude our work in Section VII.

II. Background

In this Section, we briefly introduce the background knowledge of the OpenFlow communication process, the topology management service and the northbound Rest API in SDN controllers.

A. The OpenFlow Communication Process

OpenFlow is the most widely deployed southbound interface. It defines the forwarding behavior and message format for the interaction between the OpenFlow switch and controller. Unlike traditional switches, OpenFlow switches forward packets according to the flow entries in the flow table. Each flow table in the switch contains a set of flow entries. A flow entry consists of the match fields, counters and a set of instructions that are applied to match packets [10]. When a switch receives a packet, it starts matching the flow table according to the specified match fields. If matched, the packet will be processed based on the instruction which contains in the matched flow entry. Otherwise, the packet is forwarded according to the table-miss flow entry. By default, the unmatched packets will be sent to the controller via `Packet-In` messages. The OpenFlow message plays an important role in the communication between the switch and controller. There are three types of messages: controller-to-switch, asynchronous, and symmetric. Controller-to-switch messages, such as `Packet-Out` and `Features` messages, are initiated by the controller and used to manage or control the switch. Asynchronous messages are initiated by the switch to inform the switch state change event to the controller. It includes `Packet-In` and `Port-Status` messages. Symmetric messages could be initiated by either a switch or a controller to complete the connections. Also, they are used to process some problems between switches and the controller. When the controller receives `Packet-In` messages that contain unmatched packets, the forwarding service would compute a route for these packets. But, if there no route for these packets, controller would instruct the switch flood these packets to other ports through the `Packet-Out` messages.

B. The Topology Management Service

The topology management service in the controller is the foundation of route computing. It is responsible for updating the network topology. To collect the information about the switch and monitor the link state, the controller sends link layer discovery protocol (LLDP) packets in the form of `Packet-Out` messages to connected switches periodicity. It then learns the network information according to the `Packet-In` messages and updates the network topology view simultaneously. In general, the topology management service includes *host discovery*, *switch discovery* and *link discovery*. Host discovery is passive. It learns and updates the host information (i.e., MAC address, Inport) based on the `Packet-In` messages caused by mismatched packets. The discovery of the switch occurs when the switch establishes a connection to the controller. The controller learns and stores the switch information (i.e., the port ID and datapath ID) to update the topology information. Different from the discovery of the host and switch, link discovery is proactive. The controller sends LLDP packets to the switch and computes the link based on the neighbour's Packet-In and LLDP messages. Normally, the controller provides an intuitive way to manage and monitor the network topology with a web-based graphical user interface (Web GUI). For example, the Web GUI of Floodlight controller provides a way for users to view controller state information, connected switches, inter-switch links, devices or hosts, flows installed in switch tables, and the overall network topology [11]. Most OpenFlow statistics could be queried through the Web GUI, and the results are displayed in a tabular fashion.

C. The Northbound Rest API

The core of SDN programmability is the Northbound Rest API, which provides an abstraction of the network infrastructure with a programmable interface for SDN applications to make use of the controller services and configure the network dynamically. Different from the southbound, a standard northbound API for SDN does not exist. As a result, many controllers, i.e., Floodlight, Ryu, OpenDaylight, implement their own service abstractions adopted a Rest-based approach along with data representation formats like JSON and XML. Users can easily develop northbound applications to manage the network through the features exposed by the controller. Besides, the Rest API is also important to the cloud computing applications. Floodlight controller provides a built-in virtual-network module to the application of OpenStack Quantum [12] through the exposed Rest API. Banikazemi *et al.* [13] implemented a module in Floodlight providing a Rest API for virtual networks management.

III. Packet Injection Attack

In this section, we describe the new security vulnerability in existing SDN controller. We demonstrate how attackers can inject packets to the network, spoof network service and consume the resources.

A. Threat Model

A packet injection attack can happen in SDN network where users can control one or more hosts to inject manipulated packets. The adversary does not require more privileges than regular users or need to compromise the controller's system. We assume that the attacker controls several end host servers and is able to generate packets with crafted head fields.

We also assume the target SDN controller works in the reactive mode, which is widely used by most mainstream controllers, such as Floodlight [14], Pox [15], etc. In the reactive mode, switches would generate and forward Packet-In packets for incoming unmatched packets to the controller. Note that our threat model is also consistent with previous work including *TopoGuard* [16] and *FloodGuard* [4].

We illustrate a simple scenario in Figure 1. There is at least one OpenFlow switch which receives the traffics from end hosts. The OpenFlow controller processes Packet-In messages from the switch and sends flow rules to the switch in the form of Flow-Mod messages. Such a threat model is reasonable and common since SDN is designed to replace existing networking infrastructure.

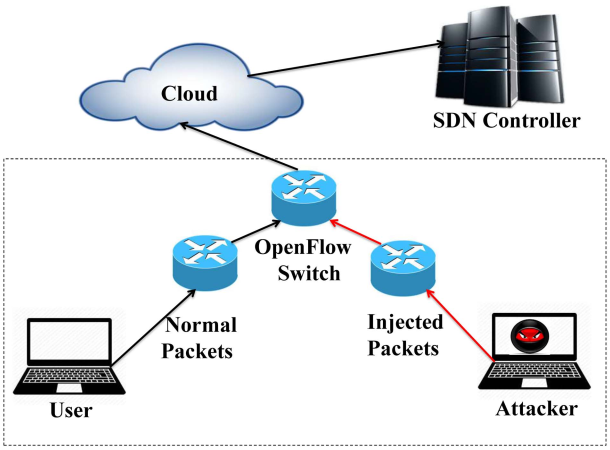


Fig. 1. Attack scenario.

B. Packet Injection Attack

As described in Section II, in the reactive SDN environment, OpenFlow switches forward unmatched packets to the controller through the OpenFlow channel. The topology management service in the controller learns the host information by monitoring the Packet-In messages and updates the topology information as well as host profiles. Also, the Rest API provides devices information to the network applications. In the process, there is no mechanism in the controller to ensure the legitimacy of Packet-In messages caused by unmatched packets. Any unmatched packets could trigger the Packet-In messages. Such a design could be exploited by attackers who control one or more end hosts to mount packet injection attacks that a large amount of fake packets are falsified. Once those falsified packets are sent to the OpenFlow switch, table-misses would occur because there are no corresponding flow rules for these packets. As a result, the switch sends Packet-In messages with the information of these packets to the SDN controller through the OpenFlow channel. Those Packet-In messages would lead to the following consequences: (1) The topology management service module would learn the host information, create a host attachment and update the topology and device Rest API information. (2) Those messages would be further processed by the forwarding service module. Since the controller does not have real destinations for those packets, it would instruct the switch to flood these packets to other ports except the inport in the form of the Packet-Out messages. (3) With the explosion of the topology information, the applications built on the top of the Rest API would suffer huge performance degradation.

To demonstrate the feasibility and effectiveness of the packet injection attack, we conduct experiments in an OpenFlow simulated environment with Mininet [9]. Mininet is a common simulation tool used in OpenFlow networks based on network namespaces. Processes can only access resources inside their namespaces [17]. To implement this attack, we create two hosts and an OpenFlow switch which is connected to a SDN controller. The experiment topology is initiated in the Mininet as depicted in Figure 1. Then, we generate

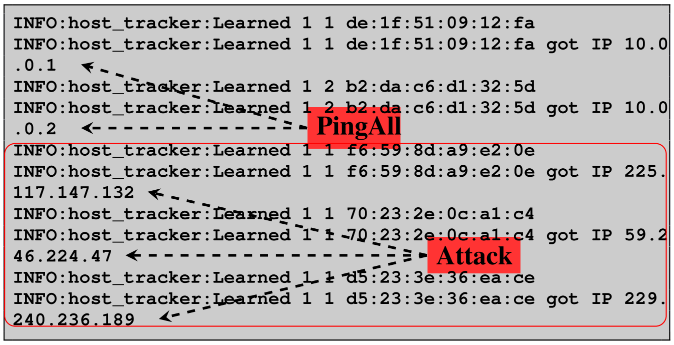

```
INFO:host_tracker:Learned 1 1 de:1f:51:09:12:fa
INFO:host_tracker:Learned 1 1 de:1f:51:09:12:fa got IP 10.0
.0.1
INFO:host_tracker:Learned 1 2 b2:da:c6:d1:32:5d
INFO:host_tracker:Learned 1 2 b2:da:c6:d1:32:5d got IP 10.0
.0.2
INFO:host_tracker:Learned 1 1 f6:59:8d:a9:e2:0e
INFO:host_tracker:Learned 1 1 f6:59:8d:a9:e2:0e got IP 225.
117.147.132
INFO:host_tracker:Learned 1 1 70:23:2e:0c:a1:c4
INFO:host_tracker:Learned 1 1 70:23:2e:0c:a1:c4 got IP 59.2
46.224.47
INFO:host_tracker:Learned 1 1 d5:23:3e:36:ea:ce
INFO:host_tracker:Learned 1 1 d5:23:3e:36:ea:ce got IP 229.
240.236.189
```

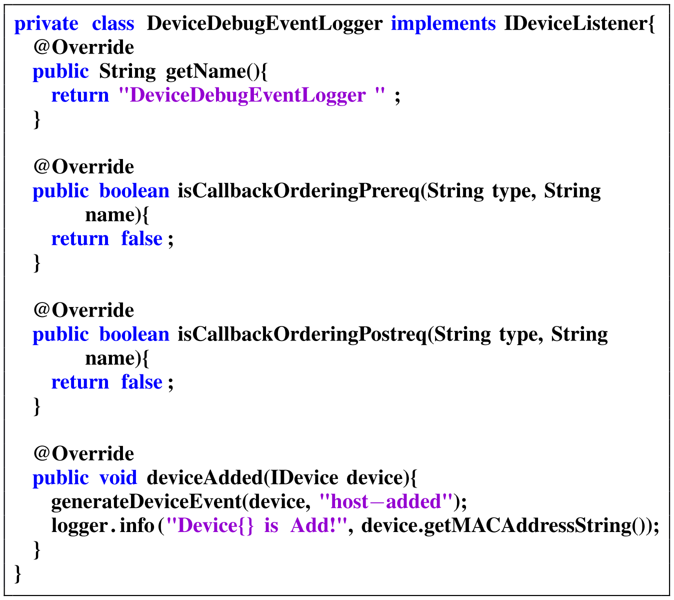
Fig. 2. Pox with and without attack.

manipulated packets and send these packets using Scapy [18] to the switch in a rate of 100 packets per second. We choose two popular controllers: Pox and Floodlight controller.

1) Spoofing: We start the Pox controller with three services: (1) The forwarding service; (2) The host tracker service; and (3) The OpenFlow discovery service. The forwarding service is responsible for installing exact-match rules for each flow. The host tracker service, as a part of topology management service, keeps tracking the host location and configuration information in the network. The OpenFlow discovery service discovers the connectivity with OpenFlow switches by sending out LLDP packets. In order to verify the functions of the host tracker service, we perform a reachability test in the Mininet environment. The result is shown in Figure 2. We find that the host tracker service can learn the location information including switch *DPID*, *Inport*, *MAC address* and *IP address* from the `Packet-In` messages. Then, we launch the attack by constructing massive packets and injecting these fake packets into the network through the host "10.0.0.1". In Figure 2, we can see that the port "1" in the switch "1" adds a lot of host devices with fake MAC addresses and IP addresses. However, these hosts are actually not existing in the network.

Different from the Pox controller, Floodlight does not display additional device information in the terminal window. In order to provide a more intuitive way to display device information, we modify the *DevicemanagerImpl.java* file. As shown in Figure 3, we add a *logger.info* function to output the device *MAC address* information. The *DeviceListener* class tracks the device information and stores it in the device object dynamically. Then, we conduct the same reachability and attack test in Mininet environment with the Floodlight controller. The result is shown in Figure 4. We can see that the device management service (similar to the host tracker service in the Pox controller) is also cheated by the `Packet-In` messages. It learns and stores the fake device information, and then provides this information to the device Rest API and network topology applications.

2) Denial-of-Service: A large amount of injected packets can not only deceive the controller service, but also launch denial-of-service attacks on networking applications which are based on these services. To demonstrate the consequences, we start the Floodlight controller and a Firefox browser with the Web GUI opened. We continually inject packets into the OpenFlow switch and refresh Web GUI on a timed interval. With the increase of attacking packets, we find that the response speed of Web GUI is slowing down. Finally, the Web GUI is even collapsed due to the large amount of device information.


```
private class DeviceDebugEventLogger implements IDeviceListener{
  @Override
  public String getName(){
    return "DeviceDebugEventLogger " ;
  }

  @Override
  public boolean isCallbackOrderingPrereq(String type, String
        name){
    return false;
  }

  @Override
  public boolean isCallbackOrderingPostreq(String type, String
        name){
    return false;
  }

  @Override
  public void deviceAdded(IDevice device){
    generateDeviceEvent(device, "host-added");
    logger.info("Device{} is Add!", device.getMACAddressString());
  }
}
```

Fig. 3. DeviceManagerImpl.

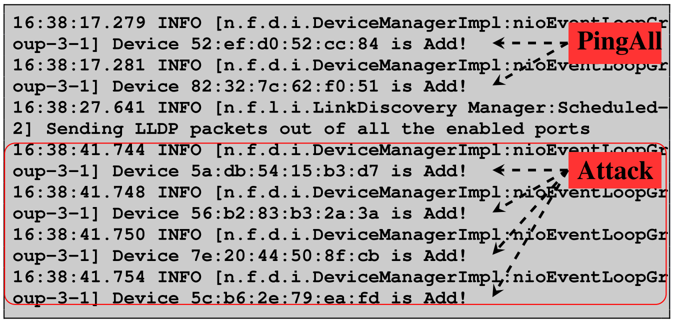

```
16:38:17.279 INFO [n.f.d.i.DeviceManagerImpl:nioEventLoopGr
oup-3-1] Device 52:ef:d0:52:cc:84 is Add!
16:38:17.281 INFO [n.f.d.i.DeviceManagerImpl:nioEventLoopGr
oup-3-1] Device 82:32:7c:62:f0:51 is Add!
16:38:27.641 INFO [n.f.l.i.LinkDiscovery Manager:Scheduled-
2] Sending LLDP packets out of all the enabled ports
16:38:41.744 INFO [n.f.d.i.DeviceManagerImpl:nioEventLoopGr
oup-3-1] Device 5a:db:54:15:b3:d7 is Add!
16:38:41.748 INFO [n.f.d.i.DeviceManagerImpl:nioEventLoopGr
oup-3-1] Device 56:b2:83:b3:2a:3a is Add!
16:38:41.750 INFO [n.f.d.i.DeviceManagerImpl:nioEventLoopGr
oup-3-1] Device 7e:20:44:50:8f:cb is Add!
16:38:41.754 INFO [n.f.d.i.DeviceManagerImpl:nioEventLoopGr
oup-3-1] Device 5c:b6:2e:79:ea:fd is Add!
```

Fig. 4. Floodlight with and without attack.

Unlike the denial-of-service attack on the Web GUI, the OpenFlow channel and network bandwidth are correlated to attack speed. More resources are required to inject more packets in a fixed time. We adjust attack speed to measure the workload of OpenFlow channel and network bandwidth. The result is shown in Figures 11 and 12. We find that when the attack speed is set with 1,600 packets per second (pps), the network bandwidth will be reduced to zero. Under such a circumstance, the network would be blocked by these packets. Meanwhile, the workload of OpenFlow channel is gradually rising. When the packet attack speed exceeds 1,600 pps, the `Packet-In` messages from OpenFlow switch would be dropped.

3) Discussion: The packet injection attack abuses `Packet-In` packets generated by maliciously crafted packets to flood the SDN. As a result, the attack can cause damage in the reactive mode. In the proactive mode, flow rules are pre-defined by the controller in the switches' flow tables. Unmatched packets would be dropped directly by

TABLE I
MAC ADDRESS AND SWITCH PORT MAPPING TABLE

Name	Type	Field
HostMap	HashMap	MAC, DPID, PORT

OpenFlow switches, and thus the `Packet-In` events would not be triggered. As a result, a naive attack has little impact on SDN in the proactive mode. We plan to investigate attacks on the proactive mode in the future.

IV. COUNTERMEASURES

To mitigate the packet injection attack, we propose *PacketChecker* to assist the SDN controller in handling fake packets. *PacketChecker* acts as a lightweight extension to existing SDN controller. We first present the overall defense mechanism. Then we describe the core algorithm for detecting the packet injection attack. Finally, we detail the design and implementation of *PacketChecker*.

A. Defense Strategy

The root cause for the packet injection attack is that the controller does not verify the validity of `Packet-In` messages. Such a mechanism allows attackers to spoof and flood packets with random MAC or IP addresses. Traditional Ethernet switches mitigate MAC flooding attacks by limiting the access on a port to users with specific MAC addresses. These MAC addresses are either manually configured or dynamically learned. It requires constant updating of the map and suffers the scalability problem.

Inspired by methods used to prevent MAC flooding attacks [19] in traditional networks, we propose a novel method, namely *PacketChecker*, to discover malicious `Packet-In` packets in SDN networks. The key idea of *PacketChecker* is straightforward: if the controller can distinguish malicious `Packet-In` messages from normal ones, it can simply drop those packets before being processed by other modules in the controller. Different from traditional networking, SDN separates the control logic from the data plane and manages the network behavior through a logically centralized controller. Such a centralized controller can deploy network policies by installing flow rules on each switch in a real time manner. As a result, *PacketChecker* does not need a manual configuration on each switch.

To verify the legitimacy of `Packet-In` messages, we propose to bind the switch port with the host's MAC address in SDN controller. In SDN network, the centralized controller knows which hosts are connected to the switch, and is able to control the switch directly. Besides, the MAC address of the host and the switch DPID are unique. The controller can limit the normal host packets from a specific switch port, and identify the injected packets through this way.

As shown in TABLE I, *PacketChecker* sets up a mapping table for the MAC address and switch port. The switch port includes the switch datapath ID and the port number where the packets come from. When the network starts, the controller collects the host's MAC address as well as the switch's port information from the `Packet-In` messages. Based on those information, *PacketChecker* creates an entry for the host's MAC address in the Mapping Table. Once the host or switch leaves the network, the controller would receive a `Port-Status` message. Based on this message, the entry with the host's MAC address or switch port would be deleted immediately. Via this approach, the table is maintained in a dynamic and real time manner. When the controller receives a `Packet-In` message, it first judges the legitimacy of the `Packet-In` message by querying the table. If the table contains the MAC addresses and port information of a packet, it means that this `Packet-In` message is legal. *PacketChecker* then forwards it to subsequent modules for further processing. Otherwise, there is a strong possibility that such a `Packet-In` message is malicious and should be dropped. In default settings, once *PacketChecker* catches a malicious `Packet-In` message, it drops the packet directly to prevent it from entering other modules. In this way, the packet injection attack could be tackled splendidly.

Note that the MAC address of a physical device is usually unchanged in the realistic networks. As a result, the defense strategy would not affect packets from legitimate devices. In the case that the MAC address of a physical device is changed (e.g., replacing a device), additional mechanisms are needed to synchronize the mapping table. The entry associated with the MAC address should be deleted in advance. *PacketChecker* could also check those updating periodically.

Although this strategy can effectively detect malicious `Packet-In` messages, it can not reduce the number of `Packet-In` messages sent to the OpenFlow channel. In order to mitigate the bandwidth saturation problem brought by the packet injection attack, we utilize the switch to reduce malicious `Packet-In` messages. Once detecting these malicious messages, the OpenFlow switch would drop the subsequent packets according to a `Flow-Mod` message generated by the controller. We control the drop action by setting a hard and idle timeout for the flow rules. Through this method, *PacketChecker* can dramatically reduce the number of `Packet-In` messages and mitigates the overhead of OpenFlow channel.

B. Detection Algorithm

The detection algorithm of the packet injection attack is presented in Algorithm 1. A table-miss in the OpenFlow switch is configured to encapsulate the packet into a `Packet-In` message, which is sent to the controller. The `Packet-In` handler in the controller processes this message according to the Algorithm 1 firstly. If the MAC address contained in this `Packet-In` message is not in the mapping table that storing the host's MAC address and switch port, this message is legitimate. We update the mapping table to include the information of this packet by writing the MAC address, switch port and DPID, and then send it to other modules. Alternatively, if the mapping table contains this MAC address, but the switch port and DPID information are not matching the data in the mapping table, we treat this message as illegal, and return a stop command, which stops the packet from sending

Algorithm 1 Detection the Packet Injection Attack

```
Input: M (Packet-In messages), T (host's MAC address and switch port mapping table), S (OpenFlow switches)
Output: C (command)
for each $M \in S$ do
  if $M.MAC \notin T$ then
    $T.MAC \Leftarrow M.MAC$, $T.Port \Leftarrow M.Port$ and $T.DPID \Leftarrow M.DPID$
    return Command.continue
  end if
  if $M.MAC \in T$ and $T.Port = M.Port$ and $T.DPID = M.DPID$ then
    return Command.continue
  else
    return Command.stop
  end if
end for
```

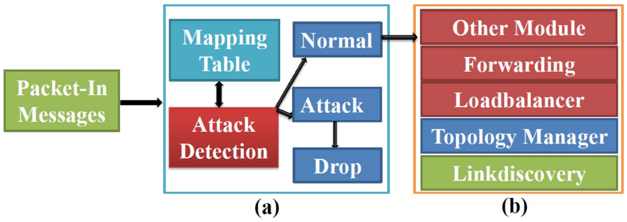


Fig. 5. The workflow of PacketChecker.

the next module in the controller. In this way, the packet injection attack could be detected in a real-time fashion. We analyze time complexity of this algorithm in section V.

C. PacketChecker

PacketChecker consists of two parts: attack detection and attack solution. The attack detection module is used to detect the packet injection attack automatically. It depends on the host's MAC address and switches port mapping table to verify `Packet-In` messages. The attack solution module is used to handle malicious `Packet-In` messages. The process is based on the execution order of the modules in the controller. The workflow of *PacketChecker* is shown in Figure 5. For the mapping table in *PacketChecker*, the MAC address and switch port information are collected by monitoring `Packet-In` messages. To maintain the mapping table dynamically, *PacketChecker* monitors the message to update the port status. When a host leaves the switch, the connected port would be shut down, and the entry in the mapping table would be deleted. Also, if the switch disconnects with the controller, the switch port in the table would be deleted.

1) Attack Detection: The `Packet-In` messages include the OpenFlow version, transaction ID, in port, Ethernet destination MAC address, Ethernet source MAC address, and so on. To verify the validity of `Packet-In` messages, *PacketChecker* extracts the source MAC address, in port and switch datapath ID. We define a class named *Port* to include the switch DPID and port number. According to the *Port*, the controller knows the location of the switch port, because the *Port* in the network is unique. Another data structure is a *Map* which implements the mapping table. The Map key is the MAC address, and the value is the *Port*. Using this *Map*, MAC address is bound with the *Port*. Then the attack detection algorithm is built on the top of those structures.

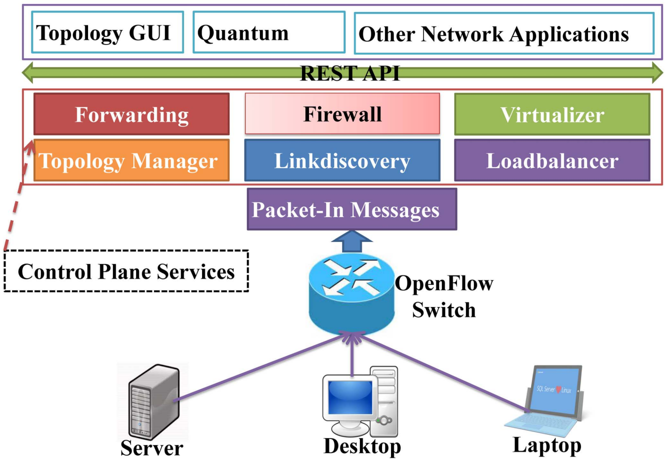


Fig. 6. Flow diagram of packet-in process.

2) Attack Solution: The attack detection module identifies the malicious `Packet-In` messages, and then sends them to the attack solution module. The attack solution module would discard malicious messages and instruct the switch to drop subsequent packets by distributing a `Flow-Mod` message. Legal messages are processed normally. As shown in Figure 6, the linkdiscovery module collects the link information firstly when SDN controller receives the `Packet-In` messages. Then, the topology management module constructs the topology using the link information, and the device management module maintains the device information. The SDN controller makes a forwarding policy based on the device and topology information. When a malicious `Packet-In` message is detected, it should be processed by attack solution module firstly. Only valid messages could be processed by the link discovery module. Malicious messages would be drop directly. Such a design allows *PacketChecker* extending the SDN controller without any modification of SDN hardware.

V. Evaluation

A. Implementation

We implement the *PacketChecker* module in the Floodlight controller and test it under the Mininet environment. The implementation of the attacking mechanisms is based on Scapy. The *PacketChecker* module implements the *IOFMessageListener* to monitor the `Packet-In` messages, and the *IOFSwitchListener* to monitor the host movement and the port status of switches. The experimental environment includes Mininet and Floodlight 1.2 running in servers with Ubuntu 15.04 and Inter Core i5-4590 3.3GHz CPU and 4GB memory.

We use Fat-Trees [20] topology to test the response time of *PacketChecker*. The Fat-tree network is widely used in data

TABLE II
THE SCALE OF TOPOLOGY

Topology	Pod	Host
1	2	4*25
2	4	8*25
3	8	16*25

center network, and it is a universal network for provably efficient communication. The scale of the topology in our experiment is shown in TABLE II. Each edge switch is connected to 25 hosts. We select H1 as the host that used to launch the packet injection attack. We utilize HttpRequester [21] to measure the response time of the device Rest API and the Web GUI of Floodlight controller which is displayed in the Firefox browser to observe the SDN network topology.

B. Evaluation

We first analyze the time complexity and measure the delay of the attack detection algorithm in our topology. Then, we test the effectiveness of the detection algorithm. Thirdly, we show the evaluation on the Web GUI which is based on the Rest API. Lastly, we launch packet injection attacks on *TopoGuard* [16] and demonstrate that *TopoGuard* cannot mitigate the packet injection attack.

1) Complexity: For the detection algorithm of the packet injection attack, the main operation is traversing the MAC address in T (host MAC address and switch port mapping table). In the worst case, it requires O(N) comparisons, where N is the number of entries in T. We use hash Map to build the mapping table. For each `Packet-In` message from a OpenFlow switch, our algorithm uses *containsKey* function to judge whether the MAC address is in the mapping table or not. If MAC address is in the table, we use *containsValue* to compare the port information from the `Packet-In` messages and judge whether it equals the value in the table. The complexity of *containsKey* function is O(1), and the complexity of *containsValue* function is O(N). Thus, we consider the complexity of the detection algorithm is O(1), in the worst case, the complexity of the detection algorithm is O(N).

2) Packet-In Process Delay: We measure the processing delay of the `Packet-In` messages to test the efficiency of the *PacketChecker* module under packet injection attacks. We send a large number of attack packets (e.g., 10,000) to the switch with an interval of 0.01s, and measure the processing delay of each packet in three scale topology by using *Java System.nanoTime* API. The API promises a precision of 1 nanosecond. The cumulative distribution function (CDF) is showed in Figure 7. We can see that almost ninety-eight percent of `Packet-In` messages processing delay are less than 20us, which is negligible for the Floodlight controller. Among three different topologies, we find that processing `Packet-In` messages have little relation with the scale of topology. When the SDN controller receives a `Packet-In` message, it would send it to the *PacketChecker* module firstly. The major delay of processing `Packet-In` messages is comparing the MAC address with the mapping table. Only in the worst case, the delay is related to the mapping table size, which is closely related to the scale of network topology. In general, the delay of `Packet-In` messages process is very small, which means *PacketChecker* can effectively handle the packet injection attack.

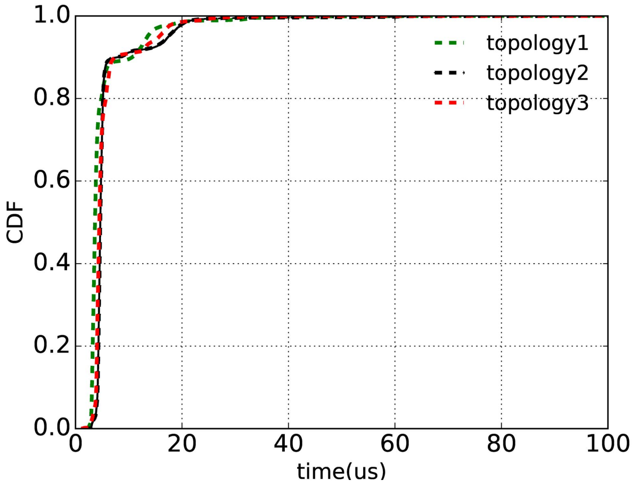


Fig. 7. CDF of measured PacketChecker delays.

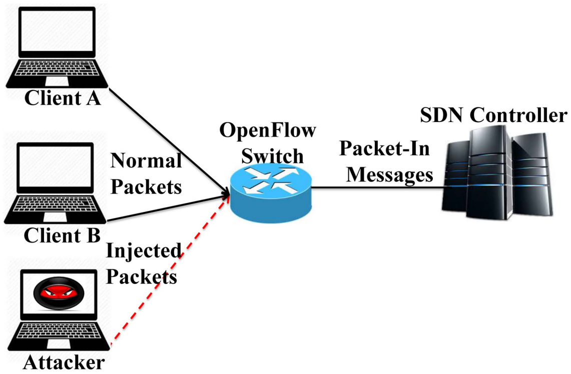
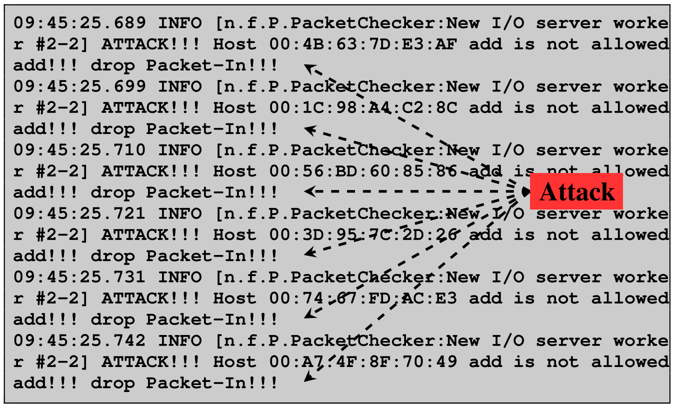


Fig. 8. Test topology.

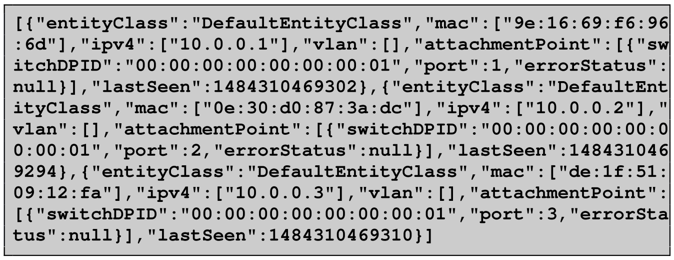
3) The Effectiveness of the Detection Mechanism: We test the effectiveness of the *PacketChecker* module from two aspects: (1) anti deception and (2) denial-of-service attack. The injected illegal packets can spoof the topology management service and Rest API, consume the bandwidth and Web GUI resource. The test environment is shown in Figure 8.

a) Anti-Deception: We measure the effectiveness by comparing the number of malicious packets detected by *PacketChecker* and injected packets sent by the attacker. The attacker keeps generating packets with different MAC and IP addresses to the switch. By counting the number of malicious packets confirmed by *PacketChecker*, we find that almost all attack packets could be detected by this module. According to the detection algorithm, the mapping table collects the right host MAC address and binds it to the switch port. Thus, packets containing other MAC addresses from the same port of the switch are considered as malicious. The detect result is shown in Figure 9. Moreover, we use RestClient [22] to access the device Rest API. In Figure 10, we find that the device Rest API only contains the Client A, B, and Attacker information. The forged host information is dropped by the



```
09:45:25.689 INFO [n.f.P.PacketChecker:New I/O server worke
r #2-2] ATTACK!!! Host 00:4B:63:7D:E3:AF add is not allowed
add!!! drop Packet-In!!!
09:45:25.699 INFO [n.f.P.PacketChecker:New I/O server worke
r #2-2] ATTACK!!! Host 00:1C:98:A4:C2:8C add is not allowed
add!!! drop Packet-In!!!
09:45:25.710 INFO [n.f.P.PacketChecker:New I/O server worke
r #2-2] ATTACK!!! Host 00:56:BD:60:85:86 add is not allowed
add!!! drop Packet-In!!!
09:45:25.721 INFO [n.f.P.PacketChecker:New I/O server worke
r #2-2] ATTACK!!! Host 00:3D:95:7C:2D:26 add is not allowed
add!!! drop Packet-In!!!
09:45:25.731 INFO [n.f.P.PacketChecker:New I/O server worke
r #2-2] ATTACK!!! Host 00:74:67:FD:AC:E3 add is not allowed
add!!! drop Packet-In!!!
09:45:25.742 INFO [n.f.P.PacketChecker:New I/O server worke
r #2-2] ATTACK!!! Host 00:A7:4F:8F:70:49 add is not allowed
add!!! drop Packet-In!!!
```

Fig. 9. Detection result of floodlight with PacketChecker.



```
[{"entityClass":"DefaultEntityClass","mac":["9e:16:69:f6:96
:6d"],"ipv4":["10.0.0.1"],"vlan":[],"attachmentPoint":[{"sw
itchDPID":"00:00:00:00:00:00:00:01","port":1,"errorStatus":
null}],"lastSeen":1484310469302},{"entityClass":"DefaultEnt
ityClass","mac":["0e:30:d0:87:3a:dc"],"ipv4":["10.0.0.2"],"
vlan":[],"attachmentPoint":[{"switchDPID":"00:00:00:00:00:0
0:00:01","port":2,"errorStatus":null}],"lastSeen":148431046
9294},{"entityClass":"DefaultEntityClass","mac":["de:1f:51:
09:12:fa"],"ipv4":["10.0.0.3"],"vlan":[],"attachmentPoint":
[{"switchDPID":"00:00:00:00:00:00:00:01","port":3,"errorSta
tus":null}],"lastSeen":1484310469310}]
```

Fig. 10. Rest API access with PacketChecker.

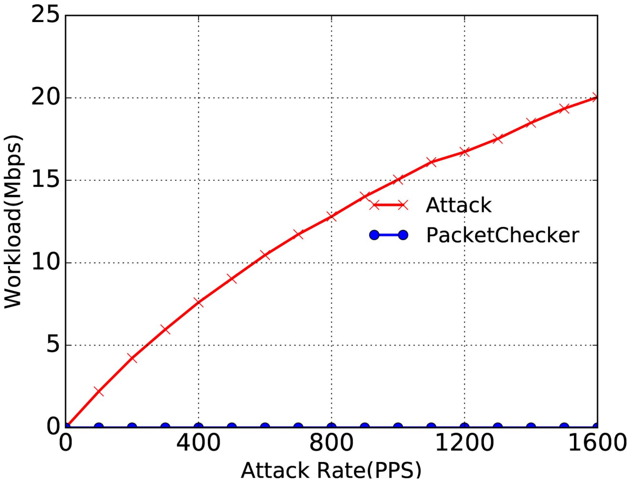


Fig. 11. Workload of openflow channel.

PacketChecker module. As we can see, *PacketChecker* module can effectively prevent the injected packets from spoofing the topology management service and Rest API.

b) Anti-DoS: We demonstrate the protection on the OpenFlow channel and bandwidth resources by *PacketChecker*. Figure 11 shows the OpenFlow channel's workload variations under the attack and defense. Without *PacketChecker*, with the increasing of attack rate, the load of OpenFlow channel increases gradually. For the *PacketChecker* module, we set a flow rule to the switch to drop the injection packets when attack is detected. We set the hard timeout and idle timeout of flow rules to 5 seconds. After that, the workload is then ignored. Dropping those injected packets in the switch can not only reduce the load of OpenFlow channel, but also save bandwidth resources. Thus it can effectively defend against the bandwidth denial-of-service attack. The result of the defense on bandwidth is shown in Figure 12. Without *PacketChecker*, the bandwidth is about 9.55Mbps in normal condition. When we start the attack and increase the attack rate, the bandwidth dramatically goes down. As the attack rate reaches 1600pps, the whole network is disabled. On the other hand, with *PacketChecker*, the bandwidth still keeps almost 9.55Mbps under attacks.

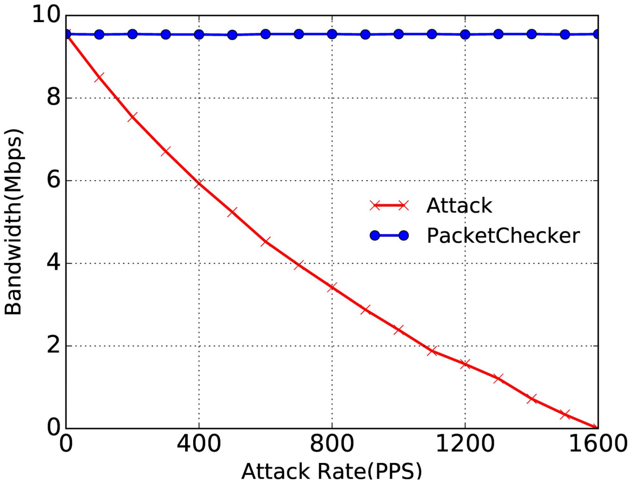


Fig. 12. Bandwidth.

In our experiment, we heuristically set the hard timeout and idle timeout of flow rules to 5 seconds. In realistic networks, the range of the timeout could be set from 0 through 65535. If the timeout is set to a relatively large value, flow rules would stay in the flow table for a long time and exhaust the space, which might cause flow table overflow. On the other hand, if this value is too small, SDN controller would send Flow-Mod messages frequently and largely increase the load on the OpenFlow channel. In our future work, we plan to design some dedicated optimization algorithms to set those timeout dynamically with the consideration of the frequency of attack, the capacity of flow table, the bandwidth of OpenFlow channel and the workload of the SDN controller.

4) Evaluation on the Web GUI: We show another protection on networking applications by testing the Web GUI. We use a Firefox browser to monitor the change of the SDN network topology based on the device Rest API. The CPU and memory usage are measured by psutil (process and system utilities) library. Psutil is a cross-platform library for retrieving information on running processes and system utilization in Python [23]. The percentage of CPU used by a process is defined as the proportion of the elapsed CPU time occupied by the task to the total CPU time. On multi-core systems, the CPU usage is the total of all cores. The topology is consisting of two client hosts connecting the OpenFlow switch, and a Floodlight controller to manage the network. The test topology is shown in Figure 8. We measure the CPU and memory usage of the Firefox process with and without our *PacketChecker* module to illustrate the effectiveness. For the attack, the rate of the packet injection is 100 packets per second. The results of the CPU and memory usage of the Firefox browser are shown in Figures 13

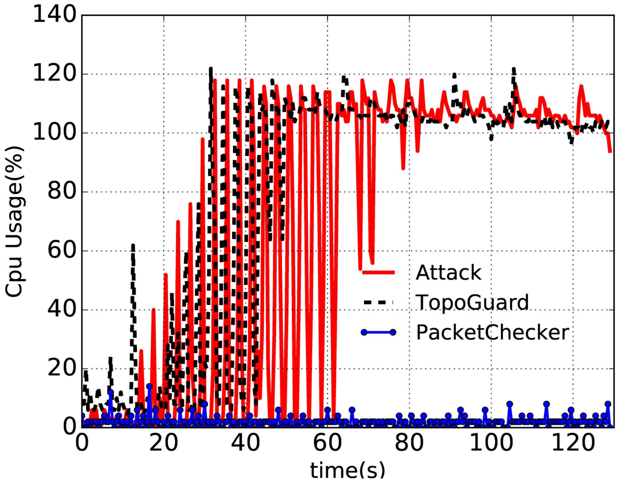


Fig. 13. CPU usage of web GUI.

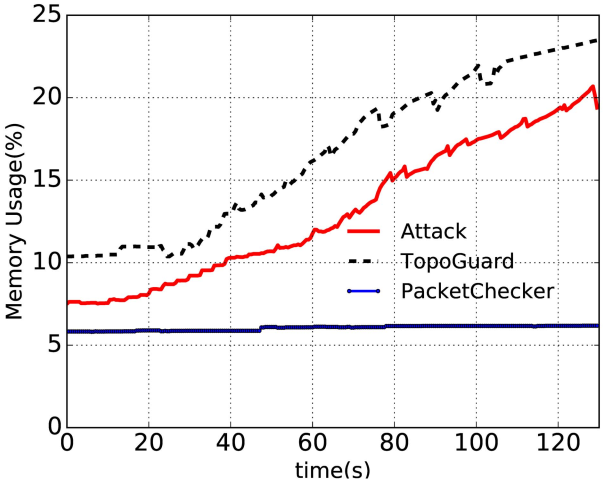


Fig. 14. Memory usage of web GUI.

and 14. The attacks occur between 20 to 120 seconds. Without the packet injection attack, the CPU and memory utilization is quite low. But after 20 seconds when we start the attack, the CPU and memory usage increase quickly. 20 seconds later, the CPU usage is over 100 and maintains in such as level in the following period of time. At last, the Firefox browser is crashed. The reason is that the packet injection attack forces the controller learning a lot of host entities. The numerous GUI read and display operations bring huge overhead to the Firefox browser and finally cause the Firefox browser being crashed. Also, during the attack, the memory utilization continues rising. That is because the increase of the host entities in the topology needs more memory to store the information and display in the Floodlight Web GUI.

Comparing to the case without the *PacketChecker* module, we can see the *PacketChecker* module can effectively defend against the packet injection attack. After the starting of the packet injection attack, the CPU and memory utilization of Firefox browser still maintains at a normal level. Since *PacketChecker* filters the malicious packets, the topology management service would not learn the large number of fake information. As a result, the CPU and memory utilization still maintains at a normal level.

5) Comparison to TopoGuard: Hong et al. proposed *TopoGuard* to protect the SDN controller from network topology poisoning attacks. *TopoGuard* verifies the legitimacy of a host migration, the origin of a LLDP packet as well as switch's port property once detecting a topology update. For `Packet-In` messages, the Port Manager in *TopoGuard* first locates the host entity in the existing Host List through the switch's port information. If the host entity is not found, the traffic is then regarded as the first-hop traffic, and the source's MAC address is recorded in the Host List.

While *TopoGuard* performs well on defending host location hijacking and link fabrication attacks, it does not verify the legitimacy of the source's MAC address. In packet injection attacks, malicious users control multiple hosts to inject manipulated packets with falsified MAC address. A new MAC address in the packets would be treated as a new host entity in *TopoGuard*. Thus, those crafted packets can pass the verification on the host migration of *TopoGuard*, and deceive the topology management service.

To validate the effectiveness of packet injection attacks and *PacketChecker*, we further launch attacks on *TopoGuard* and compare the results with *PacketChecker*. Figures 13 and 14 illustrate the experimental results, which contain the CPU and memory usage of the Firefox browser. In Figure 13, we find that the CPU usage of *TopoGuard* is very close to Floodlight without *PacketChecker*. In Figure 14, the memory usage of *TopoGuard* is about 3% more than Floodlight. This is because the port property of switch port in *TopoGuard* consumes some more memory resources. Obviously, *TopoGuard* also suffers the overwhelming of falsified packets, similar to the Floodlight controller without *PacketChecker*. The results demonstrate that packet injection attacks can also poison *TopoGuard* successfully.

VI. Related Work

Many previous studies focus on the security analysis of SDN networks and applications based on the SDN technology [24]–[28]. Shin and Gu [29] designed the CloudWatcher to monitor the large and dynamic cloud networks service. Hu *et al.* [4] developed FLOWGUARD to build robust firewalls for SDN. Matsumoto et al. presented a Fleet controller which is used to defend against malicious administrators [5]. Porras *et al.* [3] designed a security policy enforced kernel named FortNOX to detect and solve the flow rules conflict problem. All these works aim to enforce the security of SDN applications. Besides, there some other works study the scalability of SDN [30]–[32] and SDN application [33]–[36]. Different from those works, our work focus on the security problem in the data and control plane of SDN architecture.

Owing to the centralized control of SDN networks, preventing the controller from attacks is an important issue. Typical attacks on the controller include denial-of-service (DoS) attacks [37], network topology poisoning attacks and flow table overflow attacks. AVANT-GUARD [6] and FloodGuard [4] are proposed to defend against the DoS attack on SDN controller. AVANT-GUARD focuses on the inherent communication bottleneck between data and control plane as well as the speed of detection and response of the flow dynamic. Also, it mainly defeats TCP-based flooding attacks. FloodGurad is designed to defend against the data-to-control plane saturation attack. It uses a packet migration module to cache the flooding packets and send them to the controller with a rate limit in round-robin scheduling. Although it can limit the rate of `Packet-In` messages to the controller, it does not check the validity of the `Packet-In` messages, and cannot prevent the packet injection attack. Besides, Qian *et al.* [38] used an asset-centric approach to identify security threats in OpenFlow networks. They analyzed the flow table overflow attack originated from malicious applications on the controller. They compared the performance between FlowChecking and the Learning Switch on packets loss and service delaying. Similarly, a quantitative analysis of DoS attacks and mitigation mechanism is made in [39] and [40]. In order to mitigate the network poison attack, TopoGuard [16] is proposed. It is an extension to SDN controller and provides automatic and real-time detection of such attack. However, it does not consider the origin of `Packet-In` messages in solving the host location hijacking attack. SPHINX [41] detects ARP poisoning on network topology and DoS attacks on data plane forwarding.

Packet injection attack also exists in traditional network architecture. A switch-related countermeasure is proposed in [42]. Different from the traditional packet injection attack in Ethernet networks, since the OpenFlow switches do not learn packet information, such attack in SDN would lead to more disastrous consequences. We focus on the impact of packet injection attacks on data plane, control plane and propose the *PacketChecker* to mitigate this problem.

VII. Conclusion

In this paper, we unveil a new vulnerability in SDN and propose the packet injection attack. This attack can affect the topology management service, Rest API and networking applications on SDN controller. To defend against the attack, we propose and design *PacketChecker*, which is a lightweight extension to an existing SDN controller. We implement a prototype in the Floodlight controller and evaluate the effectiveness and performance of *PacketChecker*. The experimental results show that the packet injection attack can largely consume the resources of the SDN controller, and even lead to denying the service. While existing SDN controller can not defend against such attack, our design can effectively detect and mitigate the packet injection attack. Also, the *PacketChecker* module incurs negligible overhead.

Acknowledgements

We would like to thank Prof. Mauro Barni and the anonymous reviewers for their insightful and detailed comments.

References

[1] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[2] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.

[3] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Aug. 2012, pp. 121–126.

[4] H. Hu, W. Han, G. J. Ahn, and Z. Zhao, "FLOWGUARD: Building robust firewalls for software-defined networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 97–102.

[5] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from malicious administrators," in *Proc. Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 103–108.

[6] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proc. 20th ACM Conf. Comput. Commun. Secur.*, Nov. 2013, pp. 413–424.

[7] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS attack prevention extension in software-defined networks," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, Jun. 2015, pp. 239–250.

[8] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study," in *Proc. Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 165–166.

[9] Mininet. *Rapid Prototyping for Software Defined Networks*. Accessed: Aug. 2016. [Online]. Available: http://yuba.stanford.edu/foswiki/bin/view/OpenFlow

[10] *OpenFlow Specification v1.4.0*. Accessed: Oct. 2016. [Online]. Available: http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflowspec-v1.4.0.pdf

[11] *Web GUI*. Accessed: Oct. 2016. [Online]. Available: https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Web+GUI#WebGUI-Introduction

[12] OpenStack Foundation. *OpenStack Quantum*. Accessed: Apr. 2016. [Online]. Available: http://docs.openstack.org/trunk/openstack-network/admin/content/index.html

[13] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang, "Meridian: An SDN platform for cloud network services," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 120–127, Feb. 2013.

[14] *Project Floodlight*. Accessed: Jul. 2016. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[15] *Pox Controller*. Accessed: Jul. 2016. [Online]. Available: http://openflow.stanford.edu/display/ONL/POX+Wiki

[16] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2015, pp. 1–15.

[17] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. Mckeown, "Reproducible network experiments using container-based emulation," in *Proc. 8th Int. Conf. Emerg. Netw. Experim. Technol.*, 2012, pp. 253–264.

[18] *Scapy*. Accessed: Jul. 2016. [Online]. Available: http://www.secdev.org/projects/scapy/

[19] *MAC Flooding*. Accessed: Jul. 2016. [Online]. Available: http://en.wikipedia.org/wiki/MAC_flooding

[20] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, 2008.

[21] *HttpRequester*. Accessed: Oct. 2016. [Online]. Available: https://sourceforge.net/projects/httprequester/

[22] *RestClient*. Accessed: Oct. 2016. [Online]. Available: http://www.restclient.org/

[23] *Psutil 4.2.0*. Accessed: Oct. 2016. [Online]. Available: https://pypi.python.org/pypi/psutil

[24] R. Kloti, V. Kotronis, and P. Smith, "OpenFlow: A security analysis," in *Proc. IEEE Int. Conf. Netw. Protocols*, Oct. 2013, pp. 1–6.

[25] D. Kreutz, F. M. V. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 55–60.

[26] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 151–152.

[27] A. Akhunzada, E. Ahmed, A. Gani, M. K. Khan, M. Imran, and S. Guizani, "Securing software defined networks: Taxonomy, requirements, and open issues," *IEEE Commun. Mag.*, vol. 53, no. 4, pp. 36–44, Apr. 2015.

[28] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Proc. Future Netw. Services*, Nov. 2013, pp. 1–7.

[29] S. Shin and G. Gu, "CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)" in *Proc. 20th IEEE Int. Conf. Netw. Protocols*, Oct. 2012, pp. 1–6.

[30] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. 7th USENIX Symp. Netw. Syst. Design Implement.*, 2010, pp. 1–14.

[31] M. Yu, J. Rexford, M. J. Freedman, and J. Michael, "Scalable flow-based networking with DIFANE," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2010, pp. 351–362.

[32] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2011, pp. 254–265.

[33] T. Ball *et al.*, "VeriCon: Towards verifying controller programs in software-defined networks," in *Proc. ACM Conf. Program. Lang. Design Implement.*, 2014, pp. 282–293.

[34] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proc. 9th USENIX Symp. Netw. Syst. Design Implement.*, 2012, pp. 1–14.

[35] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement.*, 2014, pp. 1–15.

[36] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement.*, 2013, pp. 1–15.

[37] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, "ContainerLeaks: Emerging security threats of information leakages in container clouds," in *Proc. 47th IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, Jun. 2017, pp. 237–248.

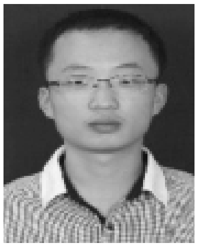
[38] Y. Qian, W. You, and K. Qian, "OpenFlow flow table overflow attacks and countermeasures," in *Proc. Eur. Conf. Netw. Commun.*, Jun. 2016, pp. 205–209.

[39] N.-N. Dao, J. Kim, M. Park, and S. Cho, "Adaptive suspicious prevention for defending DoS attacks in SDN-based convergent networks," *PLoS ONE*, vol. 11, no. 8, p. e0160375, 2016.

[40] L. Dridi and M. F. Zhani, "SDN-Guard: DoS attacks mitigation in SDN networks," in *Proc. 5th IEEE Int. Conf. Cloud Netw.*, Oct. 2016, pp. 212–217.

[41] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2015, pp. 1–15.

[42] R. Daş, A. Karabade, and G. Tuna, "Common network attack types and defense mechanisms," in *Proc. Signal Process. Commun. Appl. Conf.*, May 2015, pp. 2658–2661.



Shuhua Deng was born in 1991. He received the B.S. degree in computer science and technology from Xiangtan University, China, in 2013, where he is currently pursuing the Ph.D. degree in computational mathematics. His research focuses on software-defined networks and network security.

Xing Gao received the B.S. degree in computer science from the Beijing Institute of Technology in 2011. He is currently pursuing the Ph.D. degree with the College of William and Mary. His research interests include cloud computing, system security, and power management of data centers.

Zebin Lu was born in 1989. He received the B.S. degree in communication engineering from Xiangtan University, China, in 2013, where he is currently pursuing the Ph.D. degree in computational mathematics. His research interests are in the area of software-defined networks.

Xieping Gao was born in 1965. He received the B.S. and M.S. degrees from Xiangtan University, China, in 1985 and 1988, respectively, and the Ph.D. degree from Hunan University, China, in 2003. He was a Visiting Scholar with the National Key Laboratory of Intelligent Technology and Systems, Tsinghua University, China, from 1995 to 1996, and the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, from 2002 to 2003. He is currently a Professor with the College of Information Engineering, Xiangtan University, China. He has authored and co-authored over 80 journal papers, conference papers, and book chapters. His current research interests are in the areas of wavelets analysis, neural networks, image processing, computer network, mobile communication, and bioinformatics. He is a regular reviewer of several journals and he has been a member of the technical committees of several scientific conferences.