Securing SDN Infrastructure of IoT-Fog Networks from MitM Attacks

Cheng Li, Zhengrui Qin, Ed Novak, Qun Li, *Member, IEEE*

***Abstract*—While the IoT is making our lives much easier, managing the IoT becomes a big issue due to the huge number of connections, and the lack of protections for devices. Recent work shows that Software-defined Networking (SDN) has a great capability in automatically and dynamically managing network flows. Besides, switches in SDNs are usually powerful machines, which can be used as fog nodes simultaneously. Therefore, SDN seems a good choice for IoT-Fog networks. However, before deploying to IoT-Fog networks, the security of the OpenFlow channel between the controller and its switches need to be addressed. Since all the controller commands are sent through this channel, once compromised, the network will be completely controlled by an attacker. This is a disaster for both the network service providers and their customers. Previous works on SDN security either protect controllers themselves or make a strong assumption that the OpenFlow channel is already secured. Using TLS to encrypt the channel is not a "silver-bullet" solution due to the known TLS vulnerabilities. In this paper, we specifically investigate the potential threats of Man-in-the-Middle attacks on the OpenFlow control channel. We first introduce a feasible attack model in an IoT-Fog architecture, and then we implement attack demonstrations to show the severe consequences of such attacks. Additionally, we propose a lightweight countermeasure using Bloom filters. We implement a prototype for this method to monitor stealthy packet modifications. The result of our evaluation shows that our Bloom filter monitoring system is efficient, and consumes few resources.**

***Index Terms*—SDN, IoT, Fog Computing, MitM attack**

I. Introduction

From smart homes to smart cities, the Internet of Things (IoT) is becoming an increasingly important part of our daily lives. According to [1], there will be 12.2 billion M2M connections by 2020. Managing such a huge amount of connections is a big challenge for network administrators. Furthermore, objects in the IoT are usually resource limited. Classical computing-intensive security methods such as encryption and anti-virus software cannot be directly deployed on them. Therefore, it is necessary to secure IoT devices with the help of the network infrastructure. Under such circumstances, traditional networks are no longer suitable for the IoT. On the other hand, Software-defined networking (SDN), which brings many new features such as network programmability, centralized control and so on, enables owners to automatically manage the entire network in a flexible and dynamic way. With these benefits, many believe that the future of the IoT will be based on SDN. Therefore, several works [2] [3] are proposed for the future IoT.

Together with the IoT and SDN, fog computing is also drawing much attention. In fog computing, there are additional fog nodes between the traditional cloud and user clients. Cloud servers may offload tasks to these fog nodes and data from the clients may be cached on the fog nodes. Fog computing can be considered to be a feasible solution for an IoT implementation for several reasons. First, because the IoT generates large amounts of sensor data, sending all the data directly to the cloud is unrealistic. The fog devices, which are much closer to the sensors, can pre-process or aggregate the sensor data before sending it out to the cloud. This saves upstream network bandwidth. Second, since many IoT services are time sensitive, the cloud is not suitable for such IoT tasks due to the significant traffic latency. In this case, some lightweight processes can be migrated to nearby fog nodes, moving the computational resources closer to the IoT devices. This saves processing time. Because both SDN switches and fog nodes are relatively powerful nodes in a typical IoT deployment, they are usually combined together, which is a perfect way to integrate the functionality of both SDN and fog computing.

Though deploying IoT-Fog networks using SDN seems promising, security issues are inevitable here. Take smart home applications as an example. The fog nodes deployed in smart homes may not be configured well due to the users' lacks of expert knowledge, which may introduce vulnerabilities to the fog nodes. Furthermore, because fog nodes and SDN switches are usually combined together, vulnerabilities in fog nodes may be leveraged by attackers to compromise the SDN switches they control. Therefore, it is necessary to have security mechanisms to further monitor and enhance the security of the SDN infrastructure in IoT-Fog scenarios.

In SDN, the controller controls all the switches through "OpenFlow" channels. Commands and requests from the controller, as well as status and statistics from the switches, are transmitted through the OpenFlow channels. Therefore, the security and reliability of OpenFlow channels between the controller and switches are critical for proper SDN operation, configuration, and management. If an attacker were to intercept and/or modify the messages on these channels, he or she could send fake messages to the switches and the controllers, launching a wide variety of attacks such as denial of service (DoS), or man-in-the-middle (MitM) attacks.

OpenFlow channels, once intercepted, may bring disastrous circumstances to both the network providers and their customers. For example, an attacker can collect customers' sensitive information (e.g. sensor data depicting a user's daily behavior) by commanding the switches to send copies of packets containing such information to the attacker. In this way, sensitive user information will be leaked to attackers. With network infrastructure under such a threat, SDN has more security concerns than a traditional network. Taking another example, the attacker can send fake packets, on behalf of the switches, to the controller, poisoning the controller's global view of the network topology. With the incorrect topology, the controller may misconfigure other well-behaved switches, which may cause the network connectivity outages. The result is a horrible user experience, and substantial revenue lost. With such potential threats still viable, SDNs will never fully replace traditional networks. Even though it offers many new attractive features, without solving these problems, all the flexibility is meaningless. Therefore, work should be done to protect the OpenFlow channels from interception.

One may leverage cipher techniques to encrypt the channel after authentication. However, authentication and encryption alone cannot guarantee the safety of the OpenFlow channels. TLS, for example, is one of the most popular cryptographic protocols. However, there are still works exploiting vulnerabilities in its cipher suites and the protocol itself [4]. In [5], the attacker can compromise a TLS link by stealthily installing a client certificate. Moreover, since smart embedded devices in IoT have limited resources, some safe but computing intensive protocols cannot be deployed on them. Without secure communicating, these devices are more vulnerable to be compromised, increasing the risks of attacks against OpenFlow channel. Even assuming it were perfectly safe, fully implementing TLS is very difficult. [6] indicates that most SSL implementations are partially implemented, and contain potential vulnerabilities. Furthermore, if the attacker were to obtain the credentials or passwords of the switches or controllers via some other ways, there are limited approaches to detect and defend against the attacks. In general, we cannot only rely on cipher techniques. There should be other complimentary systems to secure OpenFlow channels. To detect such attacks, it may be possible to use a packet monitor to investigate those packets in the OpenFlow channels. However, the attacker does not necessarily change all the packets passing through the channels. With only one or two packets inserted or dropped, the attacker can easily change a switch's behavior. Therefore, monitoring the channel is not efficient. Besides, developing another monitoring system could cost much time and money.

In this paper, we mainly focus on the security issues of OpenFlow channels, especially MitM attacks. We propose approaches to launching MitM attacks on OpenFlow channels and investigate several subsequent attacks. We also implement demos for such attacks. We show that an attacker can use a small script to modify flow tables, collect information, and poison the controller's view. We also propose a countermeasure to detect MitM attacks by leveraging Bloom filter. We extend the OpenFlow protocol to incorporate our Bloom filter method and implement a prototype system which can serve as a complementary system to a variety of cipher techniques, such as TLS, to protect the OpenFlow channel from MitM attacks. Compared with standard packet monitoring systems and TLS, our system is lightweight and does not require additional hardware or maintenance. The results of our evaluation show that our system is efficient, accurate, and incurs only negligent overhead. To the best of our knowledge, our work is the first to fully investigate MitM attacks on OpenFlow channels and develop a monitoring system based on SDN for such attacks.

In summary, our contributions are as follows:

- We build demonstrations of these attacks to show how the attackers modify flow paths, collect sensitive information, and poison the controller's global view. Our implementations are relatively simple scripts with a few lines.
- Based on SDN features, we propose a lightweight countermeasure to detect MitM attacks against OpenFlow channel.
- We implement a prototype system to detect packet modification with Bloom filters based on SDN and extending the OpenFlow protocol.

II. MITM ATTACK IN OPENFLOW CHANNEL

In this paper, we assume both the controller and the switch are trusted. Both of them work correctly according to the OpenFlow protocol. The OpenFlow channel, on the other hand, is not trusted.

Figure 1 proposes one desirable SDN architecture in IoT-Fog scenario. Each IoT LAN has a gateway switch and a fog node. For efficiency concerns, the gateway and the fog node are usually combined together. The gateway switch in each IoT LAN is controlled by ISP controller. Since ISP Cloud is more secure, we argue that it is safer to put the controller in ISP cloud rather than IoT LAN. ISP offers its customers virtual machines with controller software installed, giving them rights to control their gateway switches and fog nodes. Usually, the gateway switch and controller in ISP cloud communicates in TLS. The goal of the attacker is to intercept this encrypted communication channel.

As introduced in [5], the attacker can launch KCI attack to intercept the communication channel between a client and a server by stealthily installing a client certificate at the client side. In order to successfully install client certificate at the gateway switch, the attacker needs a helper inside the LAN. Indicated by [7]–[11], there are a large amount of embedded smart devices are vulnerable to firmware updating attack, in which the attacker compromise a smart device's firmware through legitimate updating processes. If there is such a device inside the IoT LAN, the outside attacker can take control of it by launching firmware modification attack (1). Then the smart device, ordered by the attacker, installs a client certificate at the fog node (gateway), claiming that the fog node needs to use this certificate to identify itself in their future communications (2). After the gateway installs the client certificate, the outside attacker breaks the connection between the controller and the gateway (3) and performs KCI attack [5] to achieve MitM attack on the OpenFlow control channel (4). After these steps, the attacker has successfully intercepted the OpenFlow channel and take control of the gateway.

III. ATTACK DEMO

Here we introduce three attack demonstrations. In the first one, the attacker redirects flows in the data plane. The second one exemplifies how the attacker can collect information from the data plane. The last one shows how the attacker is able to poison the controller's view of the network. We present only three attack scenarios out of many others. The complete spectrum of possible attacks is currently unknown.

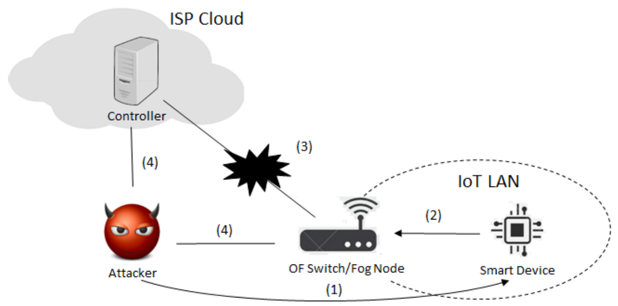


Fig. 1: Attacking model

A. *Environment Set-up*

We use Floodlight, an open source SDN controller, as our SDN controller, and use Mininet to simulate a network in our experiments. The controller and switches communicate through OpenFlow v1.3. To simplify our demos, we assume that the attacker, the controller, and the Mininet VM are located on the same local network. This assumption does not affect the result of our demos because the attacker can always intercept OpenFlow channels with spoofing techniques such as ARP spoofing. This is possible as long as the attacker exists in the path between the switch and the controller.

Since Mininet is running on a virtual machine, all simulated switches share the same IP address and remotely connect to the controller. Our attack scripts attack only the Mininet virtual machine, intercepting all simulated switches. Our configuration does not affect the final result of the demos because the technique to attack the switch's interface is identical to attacking the Mininet virtual machine.

Our attack scripts are written in Python v2.7 using the popular scapy library, which is very convenient for crafting, sending, and sniffing packets. We use this library to build fake OpenFlow commands for the switches. In our demos, we use ARP spoofing techniques to intercept the OpenFlow channel.

B. *Traffic Flow Modification*

The most straightforward attack is to stealthily modify the victim switch's forwarding table. In our experiment, the attacker blocks a certain host's traffic flow and redirects the flow to another host. Figure 2 shows the idea of this attack. The attacker inserts two OpenFlow packets, which contain flow table modification commands, into the OpenFlow channel. The first OpenFlow packet instructs the switch `s1` to modify the destination IP and MAC address of any packets originally destined for host `h4`. The new IP address and MAC address are that of host `h3`. The second OpenFlow packet commands the switch to modify the source IP address of any packets originating from `h3`, to the IP address of `h4`. As a result, if `h1` tries to communicates with `h4`, it will actually be redirected to `h3`, leaving `h1` unaware that it is communicating with a different host. To test the attack, we let `h1` ping `h4` and capture the packets transmitted using Wireshark. Figure 3 shows the packet capture results (from all the interfaces in `s1`). In the figure, the first entry shows that `s1` receives the ICMP packet from `h1` (10.0.0.1) with the destination `h4` (10.0.0.4). After being processed by the switch, the packet's destination IP address has been changed to `h3`'s (10.0.0.3) (the

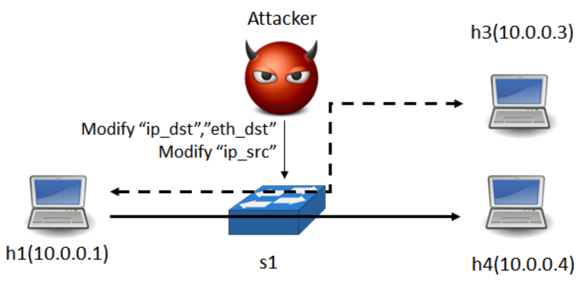
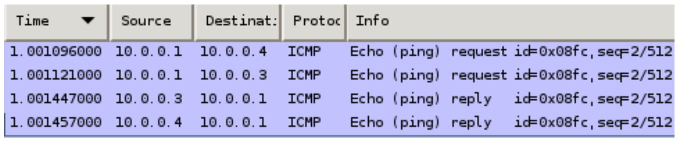


Fig. 2: Traffic redirection attack



```
Time          Source     Destinat:  Protoc  Info
1.001096000   10.0.0.1   10.0.0.4   ICMP    Echo (ping) request id=0x08fc, seq=2/512
1.001121000   10.0.0.1   10.0.0.3   ICMP    Echo (ping) request id=0x08fc, seq=2/512
1.001447000   10.0.0.3   10.0.0.1   ICMP    Echo (ping) reply   id=0x08fc, seq=2/512
1.001457000   10.0.0.4   10.0.0.1   ICMP    Echo (ping) reply   id=0x08fc, seq=2/512
```

Fig. 3: Redirection attack: packet capture result of h1 ping h4

second entry). Though not shown in Figure 3, from the reply of `h3` (the third entry), the MAC address of the packet is also changed. Passing through `s1` again, the source IP address is changed back to the IP address of `h4` (the fourth entry). These redirected paths cannot be inferred by `h1`. If `h1` is a web camera that tries to communicate with a cloud server `h4` but unexpectedly communicates with a malicious machine `h3`, all sensitive information from `h1` will be exposed to the attacker.

C. *Information Collection*

The attacker may also stealthily collect information by modifying the switch forwarding table. Figure 4 illustrates the basic idea of an information collection attack. The attacker first forges an OpenFlow packet, which contains flow table modification commands, and sends it to the victim switch. The attacker instructs the switch to send a copy of each packet targeting `h4` to the "controller", which is actually the attacker. Once the victim switch updates its forwarding table, the attacker will receive all the packets originally destined for `h4`. We let `h1` ping `h4` and again capture all packets from all the interfaces of `s1` using Wireshark. Figure 5 shows the capture result. In this demonstration, we let the attacker simply sends back the ping packet just for testing. Figure 6 shows the ending point of `h1`'s ping packets. We can see that the host receives two duplicate replies, one from `h4` and the other from the attacker. Similar as the previous demonstration, sensitive information will be leaked to the attacker, but both the client and the server will not be aware of the eavesdropper.

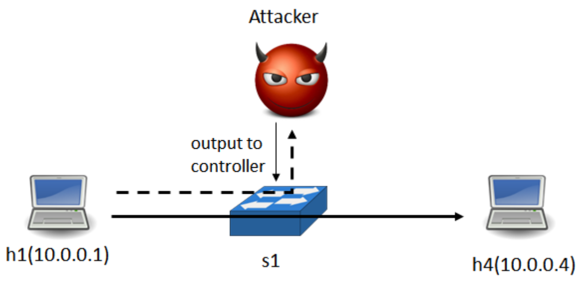
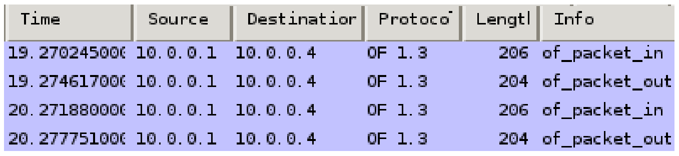
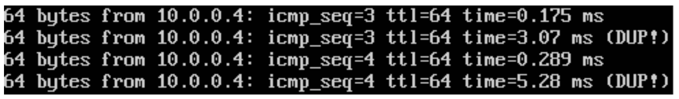


Fig. 4: Information collection attack



```
Time         Source    Destination  Protocol  Length  Info
19.27024500( 10.0.0.1  10.0.0.4     OF 1.3    206     of_packet_in
19.27461700( 10.0.0.1  10.0.0.4     OF 1.3    204     of_packet_out
20.27188000( 10.0.0.1  10.0.0.4     OF 1.3    206     of_packet_in
20.27775100( 10.0.0.1  10.0.0.4     OF 1.3    204     of_packet_out
```

Fig. 5: Information collection attack: packet capture of h1 ping h4



```
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.175 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=3.07 ms (DUP!)
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.289 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=5.28 ms (DUP!)
```

Fig. 6: Information collection attack: h1 ping h4 in terminal

D. Topology Poisoning Attack

In SDNs, the controller learns the global topology through LLDP packets. Suppose the controller commands switch `s` to output an LLDP packet through port `eth1`. Another switch `s′` receives this packet on port `eth2`. Switch `s′` includes both this packet and the port `eth2` number in a packet_in message and sends it to the controller. From this message, the controller knows that port `eth1` in `s` connects with port `eth2` in `s′`. If the attacker modifies the LLDP packets, the controller will have an incorrect view of the global topology.

Figure 7 shows the basic idea of this attack. The attacker stealthily modifies both the output port and the `max_len` field in the packet_out message. The `max_len` field indicates the maximum number of bytes the switch can send to the controller. If this field is set to 0, and the output port is set to the controller, `s1` simply ignores this message. In this way, `s2` has no chance to receive the LLDP packet, let alone forward the packet back to the controller. If the attacker does the same to `s2`, the controller will conclude that these two switches are not connected. Figure 8 shows the topology generated by the controller during the attack. Figure 8 shows the DPID of each switch. The DPID of `s1` is ”00:00:00:00:00:00:00:01” while the DPID of `s2` is ”00:00:00:00:00:00:00:02”. The third switch, which is not shown in Figure 7, is not involved in this attack. In reality, `s1` and `s2` are connected. However, the controller is fooled into thinking that they are not. If there is a packet inspection middle box along the `s1`-`s2` link, the attacker can use this method to circumvent inspection.

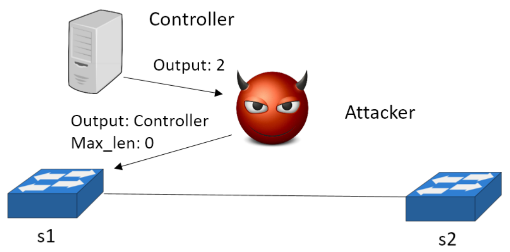


Fig. 7: Topology poisoning attack

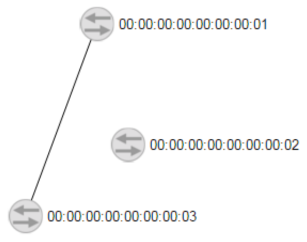


Fig. 8: Topology poisoning attack: controller view

IV. Countermeasure

In this section, a countermeasure and its OpenFlow extension to detect MitM attacks on OpenFlow channel will be proposed.

As mentioned in the previous section, the attacker can stealthily modify packets in the data plane by changing one or more switches’ forwarding table. To detect such a threat, one straightforward idea is to let the controller query all the packets that the switches forwarded, and then compare them one by one. However, this naive method will dramatically increase the burden of both the controller and the network, and also it is not efficient. To ease the burden, we propose a method to detect packet modifications using a Bloom filter. Bloom filter is a space-efficient data structure, which is used for testing the existence of an element in a set.

We let each switch along one flow locally put packets of that flow into a Bloom filter. If they put the same packets into the Bloom filter respectively, these Bloom filters should be the same. Thus, the controller can detect any packet modifications of this flow by collecting all these Bloom filters and checking the difference between these filters. If there are any differences between these filters, it is sure that the packets are modified during its delivering. Besides all the switches’ Bloom filter, we also need the origin packet sending from the sensor in case the data packets are modified at the first switch. We put a monitor process in the fog node. These processes do the same as what the switches do, putting packets from a specific flow into Bloom filters and sending Bloom filters to the controller when requested. The only difference is that these monitor processes interact with another instance in the cloud rather than the controller. Then the instance forwards the Bloom filter to the controller. The reason of using another instance is to hide the interaction between the monitor process and the controller. Because fog nodes frequently communicate with the cloud and these monitor only interact with the could when requested, the attacker has difficulties finding these monitor processes.

To apply this idea, we extend OpenFlow by adding three new message types: BF_INITIAL, BF_SUBMIT and BF_REPLY. The meanings of these messages are introduced later. Figure 9 and 10 illustrate the protocol of initializing and finalizing our Bloom filter method respectively. To start detection, the controller first sends all switches an initialization command (BF_INITIAL), which contains the following information: 1) the examined flow f, represented by matching fields used in OpenFlow; 2) a tag τ, which will be used later; 3) a set S of fields that should be omitted when computing the hash values of packets (necessary for inserting into a Bloom filter); and 4) the maximum number of packets inserted into the filter n. If n is set to 0, there is no limit for inserting packets into the Bloom filter. After receiving BF_INITIAL, each switch initializes itself according to the parameters and replies with an acknowledgment (BF_REPLY with no content) to the controller. When the controller receives a reply from every switch, it triggers the detection stage by modifying the flow table of the first switch to tag flow f with τ.

Once the controller wants to collect the Bloom filters from the switches, it first modifies the flow entry of the tagged flow f in the last switch on the path by adding a packet_in action. In this way, the controller can track the last packet of the procedure. After that, the controller commands the first switch to stop tagging flow f. When there is no packet from the last switch for a certain time, it sends out BF_SUBMIT messages to all the switches to submit their Bloom filters by BF_REPLY messages. The controller compares all the filters to find whether there is any difference among them. If any difference is found, the controller will warn the administrator about the misbehaving switches.

Limitation of the countermeasure This approach works in most cases in practice. However, in some extreme cases, for instance, all the OpenFlow channels between the controller and switches in one flow path has been intercepted, our method will not work. Besides, if the attacker modifies fields that are not in set S, our work will not work either.

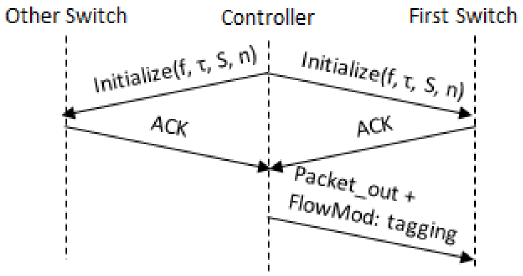


Fig. 9: The initialization of generating Bloom filter

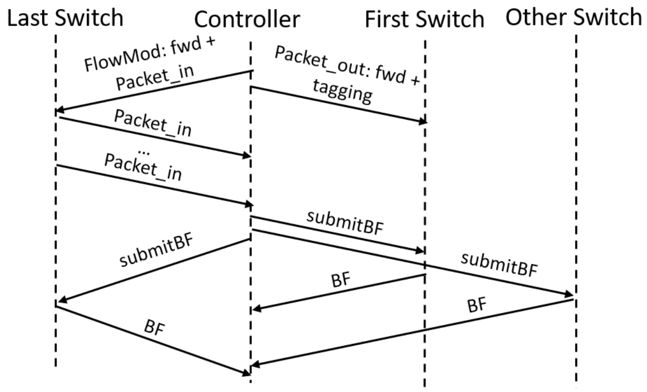


Fig. 10: The end of generating Bloom filter

V. IMPLEMENTATION

In this section, we will elaborate on the implementation of our Bloom filter monitor system, which can detect packet modifications in SDNs. Specifically, we will present the overview of the system and describe all components of the system.

A. System Overview

The monitor system, which we refer to as the "Bloom Filter Monitor System," consists of two parts. One is implemented in Floodlight controller, and the other is implemented in Open vSwitch (OVS). Figure 11 shows the architecture of our system. The controller side has one module named "Bloom Filter Monitor", which is responsible for sending out BF_INITIAL and BF_SUBMIT messages to OVS, collecting replies from

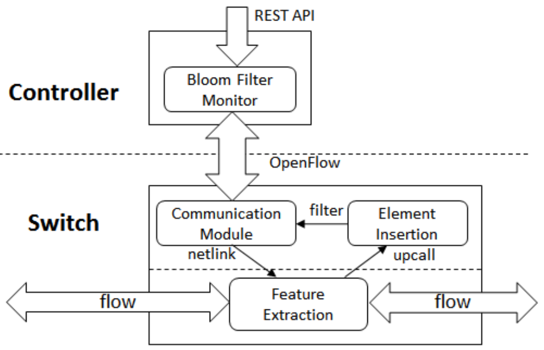


Fig. 11: Architecture of Bloom Filter Monitor System

OVS, and comparing the switches' filters. This module offers two REST APIs for administrators or other applications to conduct the Bloom filter detection phase.

The switch portion consists of two components. Generally speaking, the switch has two tasks for each packet: 1) extract examined fields (or data); and 2) insert extracted contents into the Bloom filter. In OVS, all the packets are received and forwarded in the datapath, a module that is running in kernel space where extraction starts. However, any delay inside the datapath can affect the forwarding speed. Thus, we put the hash function and Bloom filter insertion code into the user space. In this way, the switch can insert the extracted content while forwarding packets in the datapath. The switch also has one component to communicate with the controller, receiving OpenFlow messages from the controller, triggering the Bloom filter detection phase, and replying with the filled Bloom filter to the controller.

B. Controller Side Design

1) Bloom Filter Monitor Module: The main part of the Bloom Filter Monitor, as we mentioned previously, is a module in the Floodlight controller, which is automatically loaded during the initialization of Floodlight. The module has two main functions, initializing and finalizing the Bloom filter monitor method. Both of these functions can be invoked from REST APIs. The workflow of these two functions is the same as shown in Figure 9-10.

2) OpenFlow Library: To extend OpenFlow to support our new message type, we modify the source code of the OpenFlow protocol library in Floodlight. For each of our three new OpenFlow messages, BF_INITIAL, BF_SUBMIT, and BF_REPLY, one interface and several implementation classes (implemented under different OpenFlow versions) are inserted into the source code. We also change the serialization and `OFType` enum to support the serialization of these messages so that they can be transmitted through the network.

3) Floodlight Core: To enable Floodlight to handle our new messages as just another standard OpenFlow message, we modify some core codes of Floodlight. Class `OFSwitchHandshakeHandler` is responsible for receiving different types of messages and dispatching them to different components. We inserted code here to let it dispatch BF_REPLY messages to a message listener. In this way, the

Bloom Filter Monitor is able to receive and parse BF_REPLY messages from switches through a message listener.

C. Switch Side Design

1) OpenFlow Extension: To extend OpenFlow in Open vSwitch, we first insert the head structure of our three new OpenFlow messages, in the OpenFlow head files, into Open vSwitch. Then, we add new entries in enum `OPTRAW` and `OFTYPE` for our new message type. We also implement a message builder for BF_REPLY and parsers for BF_INITIAL and BF_SUBMIT, so that the Open vSwitch can understand these new messages. Finally, we add our new message handlers to the OpenFlow handler in Open vSwitch. The handler parses the message with the parser and proceeds according to the message contents. Several actions may be taken such as configuring the datapath through netlink, modifying the flow table to tag flows, and replying to the filters generated. With these modifications, Open vSwitch is able to communicate with Floodlight, which also has the OpenFlow extension.

2) Fields Extraction and Element Insertion: Open vSwitch is mainly divided into two parts: vswitchd and datapath. Vswitchd runs in the user space and is responsible for communicating with the controller and managing the flow table along with some other features. Datapath runs in kernel space and is responsible for forwarding packets. Because this part runs in kernel space, the packets can be quickly forwarded.

All the packets received by Open vSwitch first come to the datapath component where feature extraction is implemented. Once the switch receives one tagged packet, it extracts fields according to the configuration from vswitchd. After extraction, it sends the result to vswitchd using upcall, which is a mechanism used for datapath to send messages to vswitchd. In our implementation, we leverage this to send the extracted header fields to user space. Once user space receives the extracted field information, it computes the hashes and inserts them into the Bloom filter.

3) Filter Placement and Initialization: It is non-trivial to decide where to place the Bloom filter. Usually, there are several bridges inside one Open vSwitch entity. Each bridge may be connected to several different VMs. If we put the filter in the global domain, (i.e., all bridges share one filter), then the traffic flowing between VMs will not be covered. Therefore, each bridge should be treated as a switch entity and given their own Bloom filter.

In our implementation, we put the Bloom filter inside the structure `ofproto`, which is for OpenFlow protocol in OVS, since each bridge has only one such data structure, and this structure can be accessed during the processing of the upcall, where messages of extracted contents are received. When a bridge connects with the controller, it will initialize its own `ofproto` structure. The filter spaces are allocated at the same time. Once the filter has been submitted to the controller, the bridge will reset the filter for the next collection.

4) Hash Function: The hash algorithm is implemented with Murmur3 32-bit [12]. It is independent and uniformly distributed, which is apt for use in a Bloom filter. Furthermore, it is simple and efficient. For each packet, we compute the Murmur3 hashes with different seeds (to generate the k necessary hashes used in the Bloom filter) and the hash output is truncated according to the filter size. The decision of k will be discussed in next section.

VI. Evaluation

In this section, we first evaluate the performance of our Bloom filter method and delay it introduces. Then, we will test the accuracy (false positive rate) of this method.

The experimental settings are the same as the ones used in our attack demo, except that the number of switches and hosts. We use Mininet to simulate more switches and hosts to generate more traffic flows. The Floodlight controller connects with all the switches remotely. An attacker stealthily injects commands in the OpenFlow channels. There is no flow path in the data plane that consists solely of compromised switches.

A. Performance

The performance evaluation includes three parts. First, we evaluate the time cost for detecting an attack. Then, we focus on the time cost of inserting packets into filters. Finally, we investigate the introduced delays in the data plane.

1) Time cost for detecting: We only measure the time cost between the controller sending out a BF_SUBMIT message, and the controller finally finding inconsistency among the replied Bloom filters. We ignore the time interval between the BF_INITIAL and BF_SUBMIT messages since it is solely depended on the administrator.

In our first experiment, we select several flows in the data plane and let the attacker send commands to the corresponding switches to modify the destination IP address of packets (in order to keep communicating, for the purposes of the experiment, the modified IP address will be modified back by another experimenter controlled switch). To generate the selected flows, we let the two end hosts ping each other.

The number of hops along the path in the selected flows are 20, 40, 60, 80 and 100. Figure 12 shows the time interval between sending out BF_SUBMIT and when the system correctly detecting the attack. The detection time mainly contains two parts: network communication, and time for the controller to compare all the collected filters. The network communication contributes most of the delay. Additionally, we can also see that as the number of hops along the path grows, the detection time increases linearly. One more hop in the path will introduce about 0.125 ms delay in detection. Therefore, the increased rate is acceptable. The total time for detecting an attack is also relatively short. Examining flows with 100 hops takes less than 7 ms. In summary, the detection is achieved in a timely manner.

2) Time cost in OVS: As we mentioned in the previous section, our Bloom filter method has codes in both user space and kernel space. We conduct experiments to investigate the time cost in both phases, that is, the extraction and the hash computation.The experiments are repeated 30 times and the average costs are shown in Table I. From the table, we can see that the time spent in kernel is about 0.005 ms, which does not have a significant effect on the other non-selected

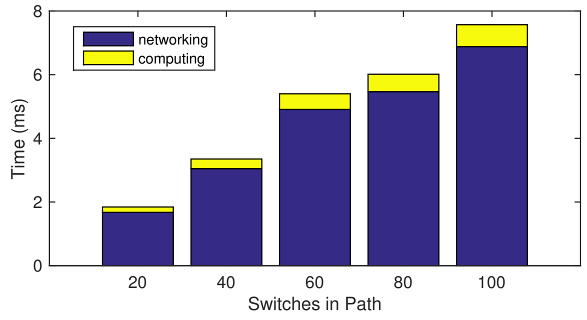


Fig. 12: Time costs for detection

TABLE I: Average time costs in each phase in OVS

Feature Extract(ms)	Hash Generation(ms)	Total(ms)
0.005	0.17	0.175

packets' forwarding. For the hash generation phase, the time measured is less than 0.2 ms, which is negligible in regards to user experience.

3) Introduced Delay: To evaluate the delay introduced by our method, we compare the round-trip time (RTT) of a ping sent between two hosts with and without our method enabled. Figure 13 shows the experiment results. We measure the RTT of flows while varying the number of hops. The RTTs under these different conditions are nearly the same. It is obvious that our method introduces negligible delays in the traffic. This makes sense since, as we mentioned in Table I, our method introduce very little delays in kernel space, and therefore has little impact on the forwarding time of other packets.

B. Accuracy

In this subsection, we evaluate the accuracy of our method. Since Bloom filters introduce false positives (but not false negatives), we only measure the false positive rate. Here, "false positive" means that some Bloom filters are actually generated from different sets of packets, but they appear to be identical. As the number of inserted packets increases, the false positive increases. This is expected because Bloom filters have a limited number of bits to store information. In this experiment, all the hosts of the network ping each other with a predetermined amount of ping packets. The experiments are repeated with different amounts of ping packets and the attacker commands the switches to modify parts of the data and destination IP addresses of all the generated flows. Switches insert ping packets flowing through them to filters and send to the controller once they receive submit requests from the controller. Based on the number of detected attacks, we compute the false positive rate. Since the size of the Bloom filter, and the number of hash functions, are two key factors

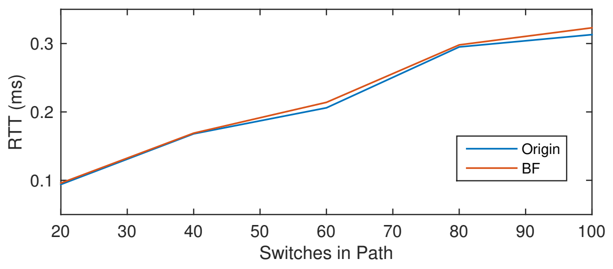


Fig. 13: Ping delay comparison

TABLE II: Parameters in accuracy evaluation

Parameter	Meaning	Default value
s	Bloom Filter size in bits	128
k	Number of hash functions in Bloom filter	5

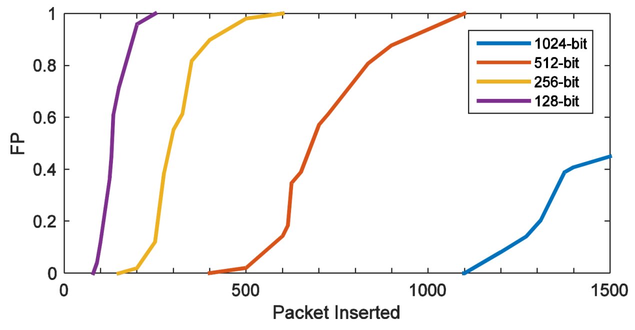


Fig. 14: False positive with different filter size

that affect the false positive rate, we mainly focusing on these factors. Table II introduces the meaning and default value of parameters in the following experiments.

1) Filter Size: We first use different filter sizes (s) to test the false positive rate with different numbers of inserted packets. Here we make $s = 1024, 512, 256, 128$ and $k = 5$. Figure 14 shows the false positive rate with different filter sizes. We can see that a larger filter can have more packets inserted into it while keeping its false positive rate at 0. For a 1024-bit filter, about 1100 packets can be inserted into the filter without introducing any false positives. However, for a 128-bit filter, if more than 100 packets are inserted, the false positive rate will dramatically increase to a very high level. Therefore, it is better to use a larger filter. However, larger filters require more space and introduce more burden to the network. Therefore, it is a trade-off between efficiency and accuracy. We use 1024-bit filters in our implementation.

2) The Number of Hash Functions: Our next experiment varies the number of hash functions (k) to investigate the false positive rate when the number of inserted packets changes. Here we make $k = 1, 2, 5, 10$ and $s = 128$. Figure 15 shows the relationship between the false positive rate, and the number of hashes used. We can see that when only one hash function is used, the filter can at most have 400 packets inserted without introducing false positives. However, for the scenario with 10 hash functions, false positives begin occurring when about 40 packets have been inserted, and it increases dramatically when more packets are inserted after this point. When many hash functions are used (k is large), the false positive rate tends to be higher as more packets are inserted into the filter.

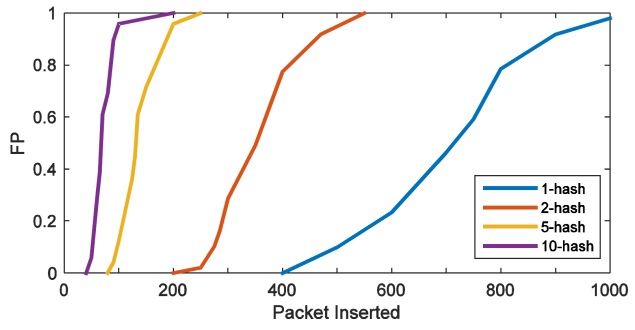


Fig. 15: False positive with different number of hashes

VII. Related Work

SDN security issues draw more attentions to researchers [13]. Seugwon et al. propose FRESCO [14], a development framework for fast developing OpenFlow security applications. In [15], a security enforcement kernel is added as an extension of Floodlight controller. Fleet [16] is another similar controller that protect the data plane forwarding from malicious administrators. [17] proposes a security invariant checking method with Yices SMT solver. All these works focus on the inner logic check of the controller rather than the misbehaved switches. With the finding of topology poising attack [18], SPHINX [19] is proposed to inspect packets from switches to the controller. Thourgh the problem it tries to solve is very similar to ours, SPHINX assumes the communication from controller to switches is trustworthy, which is not the case in our scenario.

As a promising new field, there are lots of works in Fog computing field [20]–[22]. [23] proposes a storage system for fog computing, which supports user-specified synchronization policies on the data objects. [24] presents a software architecture that eases the development of fog applications.

VIII. Conclusion

In this paper, we focus on the potential threat of MitM attacks targeting on OpenFlow channels in IoT-Fog scenario. We introduce an attack model to show how to perform such attack on our proposed SDN architecture. We also implement three attack demos to reveal how the attack works in detail. To detect such attacks, we also propose a countermeasure using Bloom filter to detect MitM attack. A prototype of this Bloom Filter Monitor is implemented by extending the OpenFlow protocol. The evaluation result shows that the Bloom filter method is both lightweight and efficient.

References

[1] Cisco, "11th annual visual networking index: Global ip traffic forecast update."

[2] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Scotch: Elastically scaling up sdn control-plane using vswitch based overlay," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 403–414.

[3] D. Wu, D. I. Arkhipov, E. Asmare, Z. Qin, and J. A. McCann, "Ubiflow: Mobility management in urban-scale software defined iot," in *IEEE INFOCOM*, 2015, pp. 208–216.

[4] Y. Sheffer, R. Holz, and P. Saint-Andre, "Summarizing known attacks on transport layer security (TLS) and datagram TLS (DTLS)," Tech. Rep., 2015.

[5] C. Hlauschek, M. Gruber, F. Fankhauser, and C. Schanes, "Prying open pandora's box: Kci attacks against tls," in *9th USENIX WOOT*, 2015.

[6] SSL Labs, "Survey of the ssl implementation of the most popular web sites," https://www.trustworthyinternet.org/ssl-pulse/.

[7] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation." in *NDSS*, 2013.

[8] K. Chen, "Reversing and exploiting an apple firmware update," *Black Hat*, 2009.

[9] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, J. Blocki, K. Fu, and D. Song, "Take two software updates and see me in the morning: The case for software security evaluations of medical devices." in *HealthSec*, 2011.

[10] C. Miller, "Battery firmware hacking," *Black Hat USA*, pp. 3–4, 2011.

[11] B. Jack, "Jackpotting automated teller machines redux," *Black Hat USA*, 2010.

[12] Austin Appleby, https://sites.google.com/site/murmurhash/.

[13] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Communications Surveys & Tutorials*, 2015.

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

[15] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software-defined network control layer," in *NDSS*, 2015.

[16] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from malicious administrators," in *ACM Proceedings of Workshop on Hot Topics in Software Defined Networking*, 2014.

[17] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *IEEE ICC*, 2013.

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

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

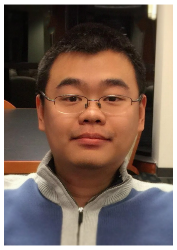
[20] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *ACM Proceedings of the 2015 Workshop on Mobile Big Data*, 2015, pp. 37–42.

[21] S. Yi, Z. Qin, and Q. Li, "Security and privacy issues of fog computing: A survey," in *WASA*. Springer, 2015, pp. 685–695.

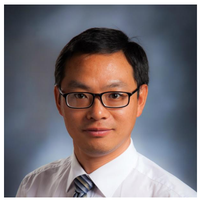
[22] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *IEEE HotWeb*, 2015, pp. 73–78.

[23] Z. Hao and Q. Li, "Edgestore: Integrating edge computing into cloud-based storage systems," in *Proceedings of IEEE/ACM Symposium on Edge Computing*, 2016.

[24] Z. Hao, E. Novak, S. Yi, and Q. Li, "Challenges and software architecture for fog computing," *IEEE Internet Computing*, 2017.



Cheng Li is a PhD candidate in the Department of Computer Science at the College of William and Mary. His research interests include SDN, NFV, machine learning and network security.



Zhengrui Qin is an assistant professor in the School of Computer Science and Information Systems at Northwest Missouri State University. He holds a PhD degree in computer science from the College of William and Mary. His research interests include cyber security and mobile computing.

Ed Novak is currently serving his first year as Assistant Professor of computer science at Franklin and Marshall College in Lancaster, PA. He recently finished his Ph.D. in May of 2016 at the College of William and Mary under the advisement of Dr. Qun Li. His research area is focused on security and privacy of smart mobile devices, and the Internet of Things.

Qun Li is a professor in the Department of Computer Science at the College of William and Mary. He holds a PhD degree in computer science from Dartmouth College. His research interests include wireless networks, IoT, edge computing, pervasive computing, and security & privacy. He received an NSF Career award in 2008.