Scientific Journal of Impact Factor (SJIF): 5.71

e-ISSN (O): 2348-4470
p-ISSN (P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 5, Issue 03, March -2018

Securing SDN infrastructure of IOT Fog computing network: A survey on Mitm attacks

Shreevidya S1, Dr.Shambavi B R2

[1]PG Student, Department of Information Science and Engineering, BMSCE
[2]Associate Professor, Department of Information Science and Engineering, BMSCE

Abstract—*In this paper, we discussed the brief overview of SDN security survey,we specifically investigate the potential threats of man-in-the-middle attacks on the Open Flow control channel, we also describe a feasible attack model in the openflowchannel, 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.*

Keywords—*MITM (Man-In-The-Middle) attacks, IOT (Internet of Things), SDN (Software Defined Networks), Fog computing networks.*

1. INTRODUCTION

Software-defined networking (SDN), which brings many new features, such as network programmability, centralized control, etc., enablesowners to automatically manage the entire network in a flexibleand dynamic way. With these benefits, many believe that the future of the IoT will be based on SDN. Therefore, severalworks [2] and [3] are proposed for the future IoT.As both SDN switches and fog nodes are relativelypowerful nodes in a typical IoT deployment, they areusually combined together, which is a perfect way to integratethe functionality of SDN. Though deploying IoT–Fog networks using SDN seemspromising, security issues are inevitable here. As fog nodes and SDNswitches are usually combined together, vulnerabilities in fognodes may be leveraged by attackers to compromise the SDNswitches they control. Therefore, it is necessary to have securitymechanisms to further monitor and enhance the securityof 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, aretransmitted through the OpenFlow channels. Therefore, thesecurity and reliability of OpenFlow channels between thecontroller and switches are critical for proper SDN operation,configuration, and management. If an attacker were tointercept and/or modify the messages on these channels, he or she could send fake messages to the switches and thecontrollers, launching a wide variety of attacks, such as denialof service or man-in-the-middle (MitM) attacks. Open Flow channels, once intercepted, may bring disastrouscircumstances to both the network providers and theircustomers. For example, an attacker can collect customers’sensitive information (e.g., sensor data depicting a user’s daily behaviour) by commanding the switches to send copies of packetscontaining such information to the attacker. In this way,sensitive user information will be disclosed to attackers. Withnetwork infrastructure under such a threat, SDN has moresecurity concerns than a traditional network. Taking anotherexample, the attacker can send fake packets, on behalf of theswitches, to the controller, poisoning the controller’s globalview of the network topology. With the incorrect topology,the controller may misconfigure other well-behaved switches,which may cause the network connectivity outages. The resultis a horrible user experience and substantial revenue lost. Withsuch potential threats still viable, SDNs will never fully replacetraditional networks. Even though it offers many new attractivefeatures, without solving these problems, all the flexibilityis meaningless. Therefore, work should be done to protect theOpenFlow channels from interception.One may leverage cipher techniques to encrypt the channelafter authentication. However, authentication and encryptionalone cannot guarantee the safety of the OpenFlow channels.TLS, for example, is one of the most popular cryptographicprotocols. However, there are still works exploiting vulnerabilitiesin its cipher suites and the protocol itself [4]. In [5], theattacker can compromise a TLS link by stealthily installing aclient certificate. Moreover, since smart embedded devices inIoT have limited resources, some safe but computing intensiveprotocols cannot be deployed on them. Without securecommunicating, these devices are more vulnerable to be compromised,increasing the risks of attacks against OpenFlowchannel. Even assuming it were perfectly safe, fully implementingTLS is very difficult. Reference [6] indicates thatmost SSL implementations are partially implemented and containpotential vulnerabilities. Furthermore, if the attacker were to obtain the credentials or passwords of the switches or controllersvia 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

complimentarysystems to secure OpenFlow channels. To detect suchattacks, it may be possible to use a packet monitor to investigatethose packets in the OpenFlow channels. However, theattacker does not necessarily change all the packets passingthrough the channels. With only one or two packets insertedor dropped, the attacker can easily change a switch's behaviour. Therefore, monitoring the channel is not efficient. Besides,developing another monitoring system could cost much timeand money.In this paper, we mainly focus on the security issues ofOpenFlow channels, especially MitM attacks. We propose approaches to launching MitM attacks on OpenFlow channels and investigate several subsequent attacks. We implementdemos for different MitM attacks. We show that an attacker can use asmall script to modify flow tables, collect information, andpoison the controller's view. We also propose a countermeasureto detect MitM attacks by leveraging Bloom filter. Weextend the OpenFlow protocol to incorporate our Bloom filtermethod and implement a prototype system which can serveas a complementary system to a variety of cipher techniques,such as TLS, to protect the OpenFlow channel from MitMattacks. Compared with standard packet monitoring systemsand TLS, our system is lightweight and does not require additionalhardware or maintenance. The results of our evaluationshow that our system is efficient, accurate, and incurs only negligentoverhead. To the best of our knowledge, this paper isthe first to fully investigate MitM attacks on OpenFlow channelsand develop a monitoring system based on SDN for suchattacks.

In summary, our contributions are as follows.

1) We build demonstrations of these attacks to show howthe attackers modify flow paths, collect sensitive information,and poison the controller's global view. Ourimplementations are relatively simple scripts with a fewlines.
2) Based on SDN features, we propose a lightweight countermeasureto detect MitM attacks against OpenFlowchannel.
3) We implement a prototype system to detect packet modificationwith Bloom filters based on SDN and extending theOpenFlow protocol.

2 OVERVIEW OF SDN SECURITY

There are clear security advantages to be gained from the SDN architecture. For example, information generated from traffic analysis or anomaly-detection in the network can be regularly transferred to the central controller. The central controller can take advantage of the complete network view supported by SDN to analyze and correlate this feedback from the network.

Based on this, new security policies to prevent an attack can be propagated across the network. It is expected that the increased performance and programmability of SDN along with the network view can speed up the control and containment of network security threats.

On the down-side, the SDN platform can bring with it a host of additional security challenges. These include an increased potential for Denial-of-Service (DoS) attacks due to the centralized controller and flow-table limitation in network devices, the issue of trust between network elements due to the open programmability of the network, and the lack of best practices specific to SDN functions and components. For example, how to secure the communication channel between the network element and the controller when operated in the same trust domain, across multiple domains, or when the controller component is deployed in the cloud?

In the past few years, a number of industry working groups have been launched to discuss the security challenges and solutions. Meanwhile, researchers have presented solutions to some SDN security challenges. These range from controller replication schemes through policy conflict resolution to authentication mechanisms. However, when the extent of the issues is compared to the degree of attention placed on them, it is clear that without a significant increase in focus on security, it is possible that SDN will not succeed beyond the private datacenter or single organization deployments seen today. The main objective of this paper is to survey the literature related to security in SDN to provide a comprehensive reference of the attacks to which a software-defined network is vulnerable, the means by which network security can be enhanced using SDN and the research and industry approaches to security issues in SDN. The paper is structured as follows: Section II provides a context to the work by introducing the characteristics of SDN. In Section III recent SDN and OpenFlowsecurity analyses are presented followed by a categorization of the potential attacks to which the architecture is vulnerable. Research work presenting solutions to these attack types is then presented in Section IV. The arrows in Fig. 1 indicate the attack categories for which solutions have been proposed and, by extension, those areas which have not yet received research attention. In Section V, the alternative view of SDN security is introduced with a survey of the research work dealing with security enhancements based on the SDN architecture. In Section VI, the two perspectives on SDN security are compared with improved functionality, open challenges, and recommended best practices identified. Section VII highlights open standards and open source industry group work on SDN security. Future research directions are identified in Section VIII. The paper is concluded in Section IX. For clarity, an overview of the Security Survey structure is presented in Fig. 1.

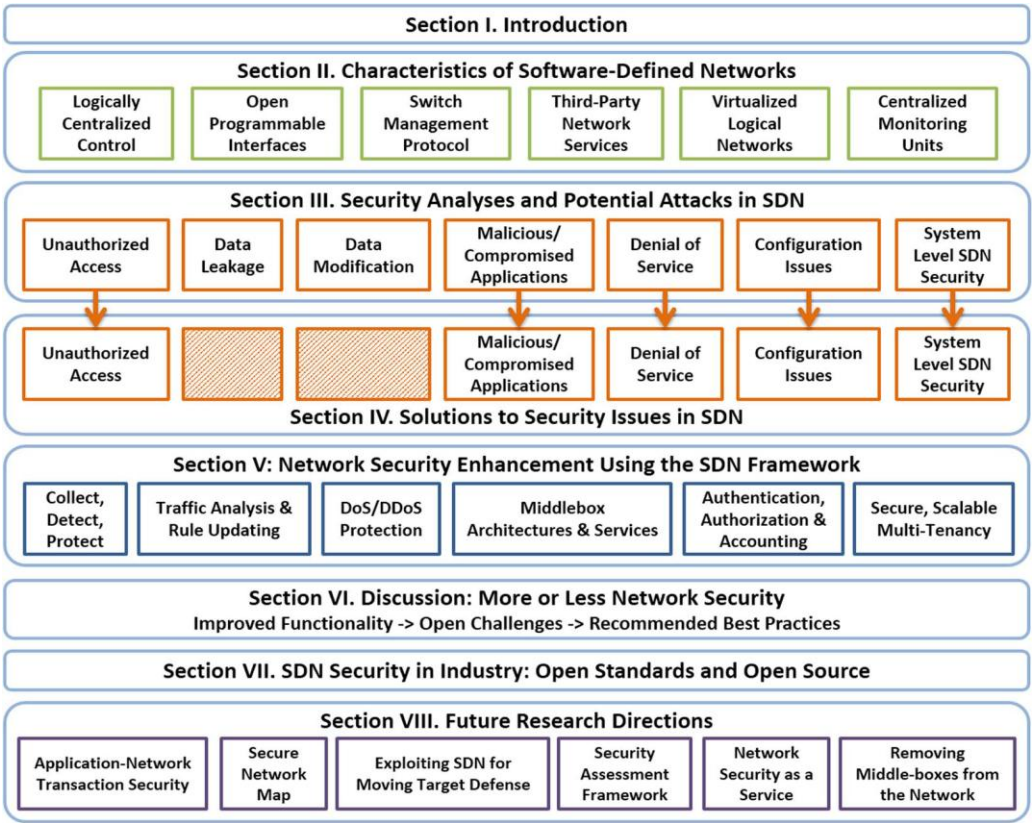

Fig 1: overview of SDN Security

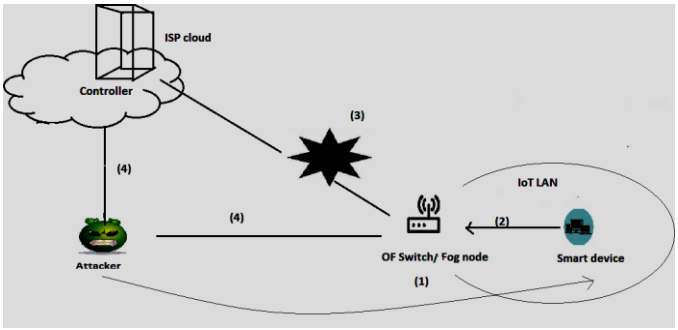

Fig 2: Attacking model.

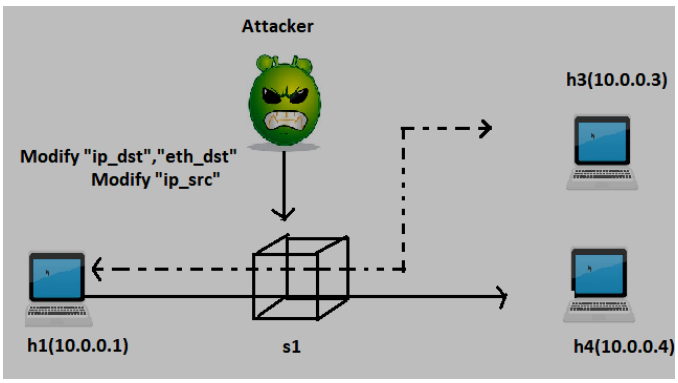

Fig 3: Traffic redirection attack.

Time	Source	Destination	Protocol	Info
1.001096000	10.0.0.1	10.0.0.4	ICMP	Echo (ping) requestid=0x08fc, seq=2/512
1.001121000	10.0.0.3	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 4:Redirection attack. Packet capture result of h1 ping h4.

3. ATTACK DEMONSTRATION

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

A. Environment Set-Up

We use Floodlight, an open source SDN controller, as ourSDN 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 Mininetvirtual machine, intercepting all simulated switches. Our configurationdoes not affect the final result of the demos becausethe 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 modifythe victim switch's forwarding table. In our experiment, the attacker blocks a certain host's traffic flow and redirects the flow to another host. Fig. 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 communicate 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. Fig. 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 second entry). Though not shown in Fig. 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. Fig. 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. Fig. 5 shows the capture result. In this demonstration, we let the attacker simply sends back the ping packet just for testing. Fig. 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.

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. Fig. 7 shows the basic idea of this attack. The attacker stealthily modifies both the output port and the max_lenfield in the packet_out message. The max_lenfield 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. Fig. 8 shows the topology generated by the controller during the attack. Fig. 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 Fig. 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.

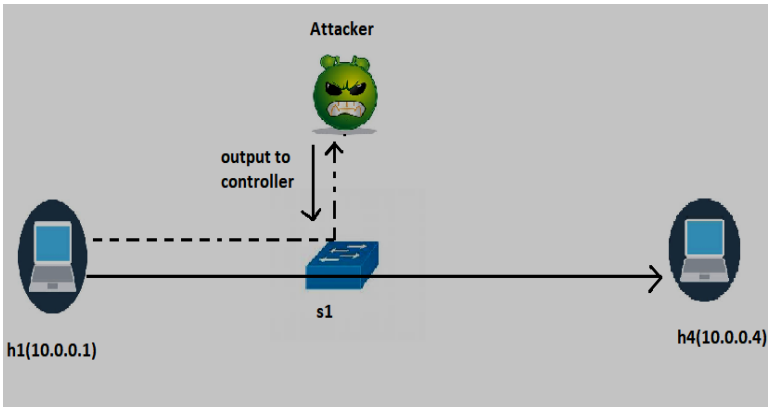

Fig: 5 Information collection attack.

Time	Source	Destination	Protocol	Length	Info
19.270245000	10.0.0.1	10.0.0.4	OF 1.3	206	Of_packet_in
19.274617000	10.0.0.1	10.0.0.4	OF 1.3	204	Of_packet_out
20.271880000	10.0.0.1	10.0.0.4	OF 1.3	206	Of_packet_in
20.277751000	10.0.0.1	10.0.0.4	OF 1.3	204	Of_packet_out

Fig: 6 Information collection attack. Packet capture of h1 ping h4


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

Fig: 7 Information collection attack: h1 ping h4 in terminal.

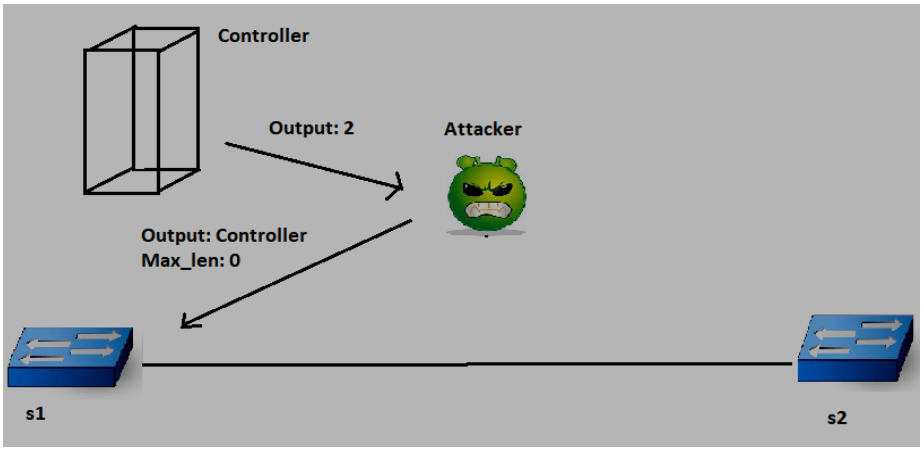

Fig: 8 Topology poisoning attack.

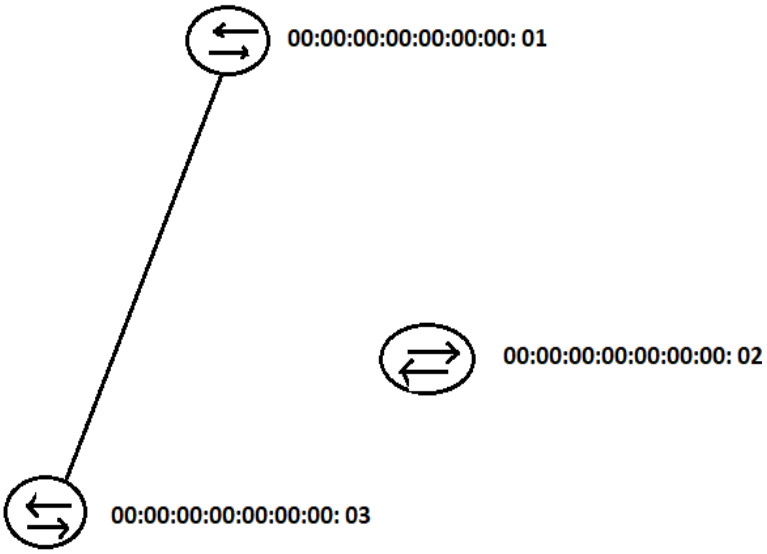

Fig: 9 Topology poisoning attack, Controller view.

4. 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. As fog nodes frequently communicate with the cloud and these monitor only interact with the cloud when requested, the attacker has difficulties finding these monitor processes. To apply this idea, we extend OpenFlow by adding three new message types: 1) BF_INITIAL; 2) BF_SUBMIT; and 3) BF_REPLY. The meanings of these messages are introduced later. Figs. 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 initializationcommand (BF_INITIAL), which contains the following information: 1) the examined flow f , represented by matchingfields used in OpenFlow; 2) a tag τ , which will be usedlater; 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_REPLYwith no content) to the controller. When the controller receivesa 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_inaction. 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_SUBMITmessages to all the switches to submit their Bloom filters byBF_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.

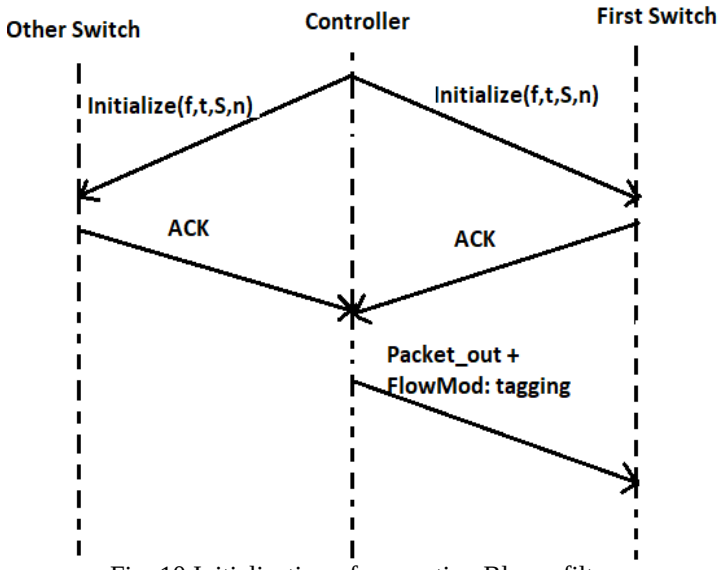

Fig: 10 Initialization of generating Bloom filter.

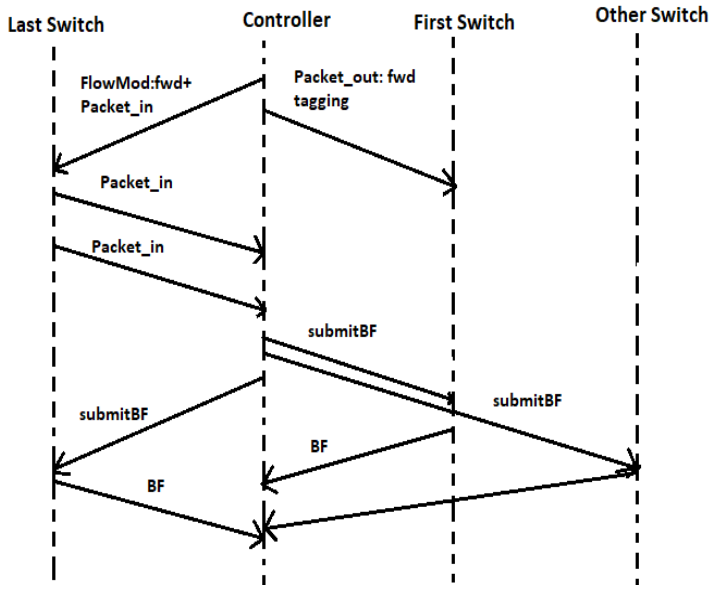

Fig: 11 End of generating Bloom filter

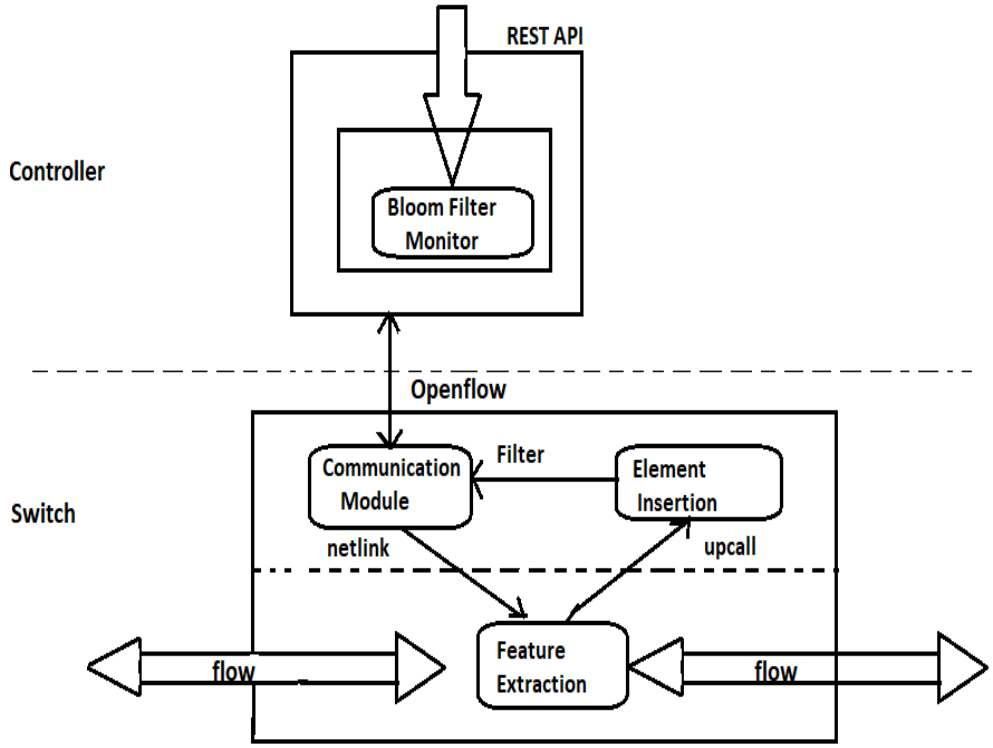

Fig: 12 Architecture of Bloom filter monitor system

A. 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*, this paper will not work either.

5. 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). Fig. 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 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: 1) initializing and 2) finalizing the Bloom filte monitor method. Both of these functions can be invoked from REST APIs. The workflow of these two functions is the same as shown in Figs. 9 and 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: 1) BF_INITIAL; 2) BF_SUBMIT; and 3) 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 OFTypeenum 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 OFSwitchHandshakeHandleris 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 OVS, wefirst insert the head structure of our three new OpenFlowmessages, in the OpenFlow head files, into OVS. Then, weadd new entries in enumOPTRAW 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 OVS can understand these new messages. Finally, we add our new message handlers to the OpenFlow handler in OVS. The handler parses the message with the parser and proceedsaccording 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, OVS is able tocommunicate with Floodlight, which also has the OpenFlowextension.

2) Fields Extraction and Element Insertion: OVS is mainlydivided into two parts: 1) vswitchd and 2) datapath. Vswitchdruns 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. As this part runs in kernel space, the packets can be quickly forwarded.
All the packets received by OVS first come to the datapathcomponent where feature extraction is implemented. Once theswitch receives one tagged packet, it extracts fields accordingto 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 userspace. 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 nontrivial todecide where to place the Bloom filter. Usually, there are several bridges inside one OVS 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 ownBloom filter. In our implementation, we put the Bloom filter inside thestructure 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 ownofprotostructure. The filter spaces are allocated at the sametime. Once the filter has been submitted to the controller, thebridge will reset the filter for the next collection.

4) Hash Function: The hash algorithm is implemented withMurmur3 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 the next section.

8. CONCLUSION

In this survey, the evidence for the two sides of the SDN security coin has been presented; that it is possible to improve network security using the characteristics of the SDN architecture,and that the SDN architecture introduces security issues.The conclusion is that the work on enhancements to network security via SDN is more mature. This is evidenced by the commercially available applications. However, research solutions have been presented to address some of the security issues introduced by SDN e.g., how to limit the potential damage from a malicious/compromised application. Work on these issues is developing encouraged by the increasing security focus of industry-sponsored standardization and research groups. 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 OpenFlowprotocol. The evaluation result shows that the Bloom filter method is both lightweight and efficient.

REFERENCES

[1] *11th Annual Visual Networking Index: Global IP Traffic Forecast Update*, Cisco, San Jose, CA, USA, 2015.
[2] A. Wang, Y. Guo, F. Hao, T. V. Lakshman, and S. Chen, "Scotch: Elastically scaling up SDN control-plane using vSwitch based overlay,"in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, Sydney, NSW, Australia, 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 *Proc.IEEE INFOCOM*, Hong Kong, 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)," IETF, Fremont, CA, USA, RFC 7457, 2015.
[5] C. Hlauschek, M. Gruber, F. Fankhauser, and C. Schanes, "Prying open Pandora's box: KCI attacks against TLS," in *Proc. 9th USENIX WOOT*, Washington, DC, USA, 2015, p. 2.
[6] SSL Labs. *Survey of the SSL Implementation of the Most Popular Web Sites*.Accessed on Apr. 2016.[Online]. Available: 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 *Proc. NDSS*, San Diego, CA, USA, 2013.
[8] K. Chen, "Reversing and exploiting an apple firmware update," in *Proc. Black Hat*, Las Vegas, NV, USA, 2009.
[9] S. Hanna *et al.*, "Take two software updates and see me in the morning: The case for software security evaluations of medical devices," in *Proc. HealthSec*, San Francisco, CA, USA, 2011, p. 6.
[10] C. Miller, "Battery firmware hacking," in *Proc. Black Hat USA*, Las Vegas, NV, USA, 2011, pp. 3–4.
[11] B. Jack, "Jackpotting automated teller machines redux," in *Proc. Black Hat USA*, Las Vegas, NV, USA, 2010.
[12] *Austin Appleby*. Accessed on Apr. 2016.[Online]. Available: https://sites.google.com/site/murmurhash/
[13] S. Scott-Hayward, S. Natarajan, and S. Sezzer, "A survey of security in software defined networks," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 623–654, 1st Quart., 2016.
[14] S. Shin *et al.*, "Fresco: Modular composable security services for software-defined networks," in *Proc. NDSS*, San Diego, CA, USA, 2013.

[15] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software-defined network control layer," in *Proc. NDSS*, San Diego, CA, USA, 2015.
[16] S. Matsumoto, S. Hitz, and A. Perrig, "Fleet: Defending SDNs from malicious administrators," in *Proc. ACM Workshop Hot Topics Softw. Defined Netw.*, Chicago, IL, USA, 2014, pp. 103–108.
[17] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *Proc. IEEE ICC*, Budapest, Hungary, 2013, pp. 1974–1979.
[18] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc.NDSS*, San Diego, CA, USA, 2015.
[19] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. NDSS*, San Diego, CA, USA, 2015.
[20] S. Yi, C. Li, and Q. Li, "A survey of fog computing: Concepts, applications and issues," in *Proc. ACM Workshop Mobile Big Data*, Hangzhou, China, 2015, pp. 37–42.
[21] S. Yi, Z. Qin, and Q. Li, "Security and privacy issues of fog computing: A survey," in *Proc. WASA*, Qufu, China, 2015, pp. 685–695.
[22] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Proc. IEEE HotWeb*, Washington, DC, USA, 2015, pp. 73–78.
[23] Z. Hao and Q. Li, "EdgeStore: Integrating edge computing into cloudbased storage systems," in *Proc. IEEE/ACM Symp. Edge Comput.*, Washington, DC, USA, 2016, pp. 115–116.
[24] Z. Hao, E. Novak, S. Yi, and Q. Li, "Challenges and software architecture for fog computing," *IEEE Internet Comput.*, vol. 21, no. 2, pp. 44–53, Mar./Apr. 2017.

