How to Detect a Compromised SDN Switch

Po-Wen Chi*, Chien-Ting Kuo*†, Jing-Wei Guo‡ and Chin-Laung Lei*
* Department of Electrical Engineering
National Taiwan University, Taipei, Taiwan
Email: {d99921015,d98921027,cllei}@ntu.edu.tw
† CyberTrust Technology Institute
Institute for Information Industry, Taipei, Taiwan
Email: ctkuo@iii.org.tw
‡ Smart Networks System Institute
Institute for Information Industry, Taipei, Taiwan
Email: jwguo@iii.org.tw

***Abstract*—SDN is a concept of programmable networking. A network manager can process each network flow through software programs. There is a hypothesis that all switches are trusted and follow programmed commands to handle packets. That is, once a switch is compromised by an attacker and does not follow the order of the network manager, this will bring a huge network disaster. In this paper, we define some attack models through compromised switches and design a detection mechanism to find these compromised devices. We evaluate our mechanism and discuss some future works.**

I. Introduction

Software Defined Networking (SDN) is a concept of programmable networks which was first proposed in [1]. A network manager can control packet processing functions in the network through his own programs. This feature enables network managers and researchers to apply their own ideas and new functions on their networks easily. Therefore, SDN becomes a very important network technology and brings a transformational paradigm shift from legacy networks.

However, SDN is based on an important assumption: all network devices will follow the network manager's commands. That is, all network switches should process packets according to programmed rules. However, once an SDN switch is compromised by an attacker, the assumption fails and the attacker will bring serious problems to the whole network. For example, the network manager may program one switch to forward some kind of packets from port 1 to port 2. After being compromised, the attacker may command the switch to duplicate this kind of packets to all ports. This will cause storming on the network and will bring lots of additional traffic burdens. Another example is the load balancing case. Many people use SDN to distribute workloads across multiple servers by different policies. An attacker may program the compromised switch to change the policy and may put most workloads on weaker servers. This will reduce the availability of servers. In these attack scenarios, attackers make switches not follow the controller's commands and the network will not operates as the manager's wish. Moreover, an attacker may find the worst case of the network manager's logic and may apply the case to the network through compromised switches. To avoid these attacks, it is necessary to develop a compromised SDN switch detection mechanism to find out black sheep from all SDN switches.

Though the security issue is taken into account when SDN was developed in the first place, most SDN specifications focus on the security channel between SDN controllers and SDN switches. Like the OpenFlow specification, it uses SSL/TLS to protect channels between OpenFlow controllers and OpenFlow switches. Data confidentiality and mutual authentication can be achieved through SSL/TLS channels. However, SSL/TLS cannot be used to detect compromised switches. An attacker can get all secrets of a compromised switch and can pass the authentication process. Therefore, we need an additional compromised switch detection mechanism for an OpenFlow network. In this paper, we design an online compromised OpenFlow switch detection mechanism. Our detection mechanism is directly implemented on the OpenFlow controller without additional devices. Besides, the network manager does not need to isolate or shut down a switch for detection. Instead, the manager can run the detect process when the network is running. Therefore, we claim our detection mechanism is very simple and practical.

In section II, we briefly introduce background knowledges about SDN technologies and related researches about compromised node detection. Our detection mechanism is proposed in section III. In section IV, we evaluate our mechanism with time cost and probability analysis. Finally, we will give some conclusions and our future research topics.

II. Background

A. OpenFlow

OpenFlow [2] is the most widely adopted implementation of SDN. It was first proposed by N. McKeown et al. in [1] and its specification is now maintained by Open Networking Foundation (ONF). It defines a new architecture that a network is managed by a logically centralized controller. It also defines how an SDN controller communicates with SDN switches, which is called the Southbound API. Each SDN switch processes packets according to its flow entries, which define packet characteristics and respective actions. Through the Southbound API, the controller can decide network behaviors by managing flow entries of all underlying switches. There are other Southbound APIs, like OpFlex[3]. However, in this paper, we just focus on the OpenFlow architecture and its protocol.

978-1-4799-7899-1/15/$31.00

B. Ryu

Ryu[4] is a component-based OpenFlow controller framework. Ryu supports fully OpenFlow version 1.0, 1.2, 1.3, 1.4. Ryu is developed in Python. It uses an event queue for recording all OpenFlow events and then dispatches these events to pre-registered handling functions. Therefore, to write a Ryu APP, one should develop packet processing functions and hook them to the Ryu platform. In this paper, we develop our detecting algorithms as two APPs on the Ryu platform.

C. Related Works

There are many security research works on SDN security issues. For example, [5], [6], [7], [8] built their DDoS mitigation algorithms on SDN networks. [9], [10] used SDN to detect abnormal traffics. [11] made use of SDN to construct a fine-grained flow-based network access control system. Adam et al. [12] proposed a forensic method for large-scale data center networks by SDN. These works provide security networks against attacks from hosts or Internet. Just like other SDN works, these works rely on the SDN architecture. That is, they need all switches are honestly controlled by their algorithms on the controller. Once SDN switches are compromised and do not follow commands from the controller, these algorithms will collapse and lose their functionalities.

In 2013, Yu et al. [13] proposed an improved authentication mechanism for protecting the SDN architecture. In this research, they deployed the PKI authentication structure on hierarchical SDN controllers and SDN switches. This work provided network resilience when some controllers and switches are known as being compromised. However, they did not propose an approach how to find compromised nodes. Moreover, a compromised switch still has the secret and undoubtedly be able to pass any authentication system with same privilege right. In 2014, Markku et al. [14] addressed problems that could be caused by compromised switches in SDN networks. For example, a compromised SDN switch may launch eavesdropping attacks, man-in-the-middle attacks and topology spoofing attacks to the network. Though Markku et al. also proposed some mitigation methods, they claimed that it is nearly impossible to detect compromised switches.

To solve the compromised SDN switch problem, we deploy Security Information and Event Management (a.k.a., SIEM) technology on SDN networks. SIEM technology can provide real-time analysis of security alerts for network managers. Therefore, we need an online detection mechanism for finding out suspicious SDN switches and generating security alerts to SIEM. In this paper, we develop two detection algorithms for catching suspicious switches and report results to the SIEM system for further processing.

III. Compromised OpenFlow Switch Detection

In this section, we will describe how to detect a compromised OpenFlow switch in an OpenFlow network. First, we will define what a compromised OpenFlow switch is and our assumptions in section III-A. In section III-B, we will show our detection ideas and some implementation details.

A. Definition and Assumption

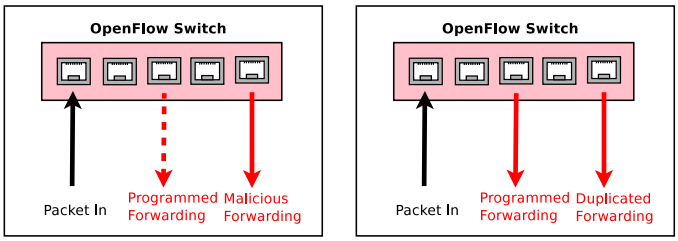
When an OpenFlow switch is compromised, the attacker may launch two kinds of attacks, passive attacks and active attacks. It is hard to detect a passive attack since the compromised switch acts as a normal switch except that the attacker can silently get all information of the switch. As for active attacks, the attacker will command the compromised switch to do abnormal actions instead of actions programmed by the controller. In this case, though the OpenFlow controller can use an *ofp_flow_stats_request* to get all flow rules from the compromised switch, the switch can easily response the correct answer to the controller without following these rules. So *ofp_flow_stats_request* cannot be used as a detection approach. In this paper, we focus on active attacks. An attacker may make a compromised switch do the following malicious actions:

- **Incorrect Forwarding**. The attacker may command the compromised switch to forward the packet without following programmed flow rules. For example, the controller programs the switch to forward a packet from port 1 to port 2 but the attacker commands the switch to forward the packet to port 3. This will break the whole switching system, like causing infinite traffic loops. Another example is that the attacker can duplicate the packet to both port 2 and port 3. This will bring additional burden on networks. Figure 1a and figure 1b are these kinds of attacks.
- **Packet Manipulating**. The attacker may modify packets that pass the compromised switch. Though packets may be protected by encryption techniques, this attack can block all communications by decryption errors. Figure 1c is this kind of attacks.
- **Malicious Weight Adjusting**. OpenFlow provides a special table called group table. A packet which is handled by the group table may be processed by one bucket in the group. The selection algorithm is based on bucket weights. This function is often used to achieve the load balancing feature. In this case, the attacker may change bucket weights and may put most pressure on some victim servers. Figure 1d is this kind of attacks.

All above attacks aim at flow actions since an OpenFlow switch processes packets according to programmed flow rules. Undoubtedly, an attacker may command the compromised switch to do more malicious behaviors other than the above attack models. We leave more detection mechanisms as our future works.

There are some assumptions in this work:

- We assume that the network is a pure OpenFlow network. That is, all switches support the OpenFlow protocol and are controlled by one OpenFlow controller.
- In our design, we use existing switches as our detection tools to test a suspicious switch. We assume that there is no collaboration between compromised switches. This means that all switches, including compromised switches, follow the controller's order to run the detection process to the under-detecting



(a) Incorrect Forwarding. (b) Duplicated Forwarding.

(c) Packet Manipulating. (d) Malicious Weight Adjusting.

Fig. 1: Attack Models.

switch. Though this assumption sounds strong, it is somewhat practical since our detection approach hides some probe packets in normal traffics. It should be difficult for an attacker to distinguish probe packets from other packets and therefore, the attacker should process packets as a normal switch. We will try to weaken the assumption in our future work.

B. Detection Mechanism and Its Implementation

In this subsection, we describe our detection design in detail. Our detection approach is to see if there is any switch that does not follow programmed flow rules to process packets. If a switch does not correctly process packets according to its rules, a network manager will reasonably doubt that the switch is compromised. Since there are many switches and each switch may have lots of rules in a network, it is impractical to detect every flow entry on every switch. Therefore, we design our mechanism as a periodic sampling detection method. The sampling approach is to randomly select a set of flow rules from random selected switches. Then the network manager will check if these selected rules are executed correctly.

Now we see how to detect if a flow rule is correctly executed. According to the attack models defined in the last section, we design two detection algorithms: **Forwarding Detection** and **Weighting Detection**. Both these two algorithms are implemented as OpenFlow APPs on the OpenFlow controller. We emphasize that both these two algorithms need no extraordinary devices and just use existing network switches. Moreover, our algorithm can be executed online without the network shutdown.

1) Forwarding Detection: This algorithm is used to check if packets are correctly forwarded. Our idea is to construct an artificial packet for the under-detecting flow entry. The algorithm then traces the packet's forwarding path and see if its path is same as programmed. The packet is embedded

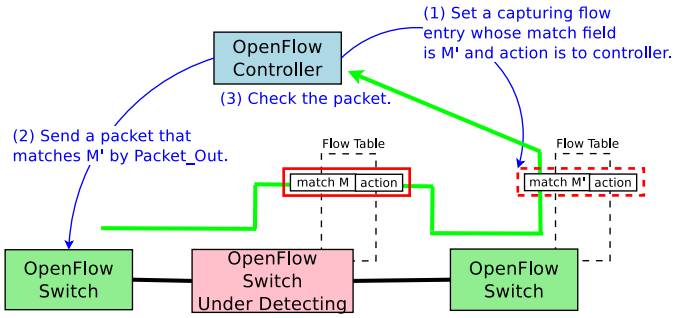


Fig. 2: Forwarding Detection.

in the network flow and is hard to be extracted. The idea is presented in figure 2.

The detection process is as follows. First, the algorithm is given an under-detecting flow entry f on a switch S where the entry match field is M. The algorithm will generate a capturing flow entry with a match field $M' = M \cup M_t$[1], where M_t is a match field additional to M and there is no packets in the network satisfying M_t. For example, if M is TCP_SRC=80 and TCP_DST=5000 and this network does not use VLAN, the algorithm can set M_t as VLAN_VID=2. Note that it is not hard to choose M_t since in a programmable network, the controller should have all network information. Therefore, the algorithm can easily pick one unused match field condition in the network as M_t. By the definition of M_t, there are two properties here:

- A packet that satisfies M' must satisfies M.
- There is no packet that satisfies M' in the network.

The algorithm constructs a flow entry f and sets its match field to M' and its output to *Controller*. Then the algorithm adds the capturing flow entry f to all neighbor switches, including physical neighbors and logical neighbors. The algorithm generates a packet p that satisfies M'. The algorithm makes use of the *Packet_Out* command to send p from one neighbor switch [2] to S. If f is correctly executed, p will be forwarded to right destination switches and therefore, these switches will relay the packets to the controller by pre-setup capturing rules. The algorithm finally checks if p comes from correct switches and if the content of p is not spoofed. By the properties of M', the controller will not additional receive packets. So this task will not bring too much traffic burden to the controller. The pseudo code of this algorithm is listed in algorithm 1.

2) Weighting Detection: This algorithm is used to check if an OpenFlow switch dispatches packets in the programmed ratio. The idea is similar to the Forwarding Detection algorithm. First, the algorithm setups capturing flow entries to all destination switches. Then the algorithm generates some artificial packets to the under-detecting switch. Finally, the algorithm collects received packet numbers of all capturing

[1]Note that OFPXMT_OFB_IN_PORT and OFPXMT_OFB_IN_PHY_PORT conditions of M should be removed from M'.

[2]This neighbor selection must satisfies OFPXMT_OFB_IN_PORT and OFPXMT_OFB_IN_PHY_PORT if necessary.

Algorithm 1 Forwarding Detection

Require: An under-detecting flow f with match field M and output set O. f is in switch S.

1: Let $S_1, \ldots, S_n$ be switches that are physically or logically connected to S.
2: Generate a match field $M' = M \cup M_t$ where M_t is a packet condition that there is no packets in the network satisfying M_t.
3: **for** S_i in $S_1, \ldots, S_n$ **do**
4: Create a flow entry f_i: match field is set to M' , output is set to *Controller*.
5: Set f_i to S_i.
6: **end for**
7: Use *Packet_Out* to send a packet p that satisfies M' from some switch to S.
8: **if** The controller receives $|O|$ packet(s) from all switches in O and each packet is equal to p. **then**
9: **return** S may not be compromised.
10: **else**
11: **return** S is compromised.
12: **end if**

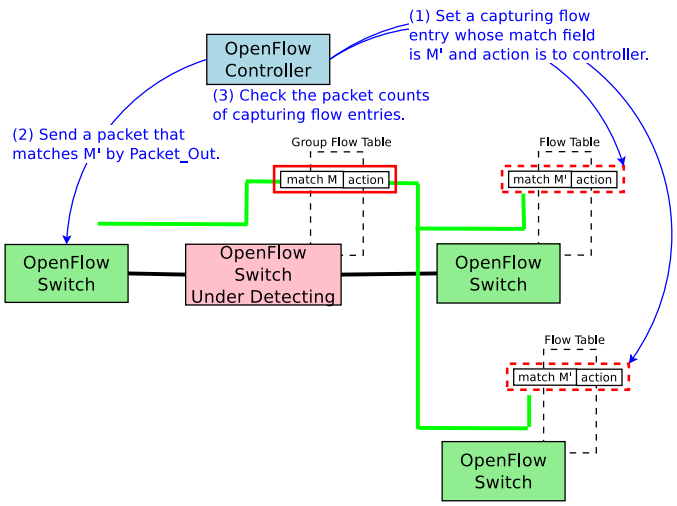


Fig. 3: Weighting Detection.

Algorithm 2 Weighting Detection

Require: An under-detecting dispatching flow f with match field M and output set O. f is in switch S. A tolerance ratio δ and detecting packet number m.

1: Let $S_1, \ldots, S_n$ be switches that are possible destinations according to f. The S_i weight in f is w_i and $w_1 + \ldots + w_n = 1$.
2: Generate a match field $M' = M \cup M_t$ where M_t is a packet condition that there is no packets in the network satisfying M_t.
3: **for** S_i in $S_1, \ldots, S_n$ **do**
4: Create a flow entry f_i: match field is set to M' , output is set to *Drop*.
5: Set f_i to S_i.
6: **end for**
7: Use *Packet_Out* to send m packets that satisfy M' from some switch to S.
8: **for** S_i in $S_1, \ldots, S_n$ **do**
9: Get received packet count c_i of f_i.
10: **if** $|c_i - m \cdot w_i| > \delta \cdot m$ **then**
11: **return** S is compromised.
12: **else**
13: continue.
14: **end if**
15: **end for**
16: **return** S may not be compromised.

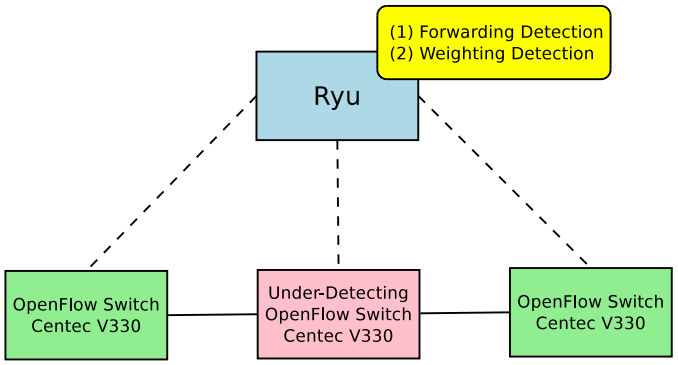


Fig. 4: Experiment Environment.

flow entries. If all received packet numbers of capturing flow entries are close to their expected values, the dispatching flow entry is treated as a normal entry. Otherwise, the entry is doubted to be compromised. The idea is presented in figure 3.

Unlike the Forwarding Detection algorithm, the capturing flow entry does not need to send packets back. Instead, it just drop captured packets. This is because each flow entry has a counter to maintain received packets and received bytes. So the algorithm can get counter information directly of each capturing entry for ratio calculation. Note that this algorithm does not check packets' contents since we leave these checks to the Forwarding Detection algorithm. Besides, this algorithm needs to send more than one artificial packets through the *Packet_Out* command. Undoubtedly, this algorithm takes more time than the Forwarding Detection algorithm but this is unavoidable. We use a tolerance ratio δ as the decision criteria. The pseudo code of this algorithm is listed in algorithm 2.

IV. Evaluation

In this section, we evaluate our detection mechanism in two ways, Detection Execution Time and Detection Probability Analysis.

A. Detection Execution Time

We implement our two detection algorithms as two OpenFlow APPs on the Ryu platform. Ryu is run on a physical machine with Intel I5 2.5GHz CPU and 8GB memory. We use three Centec V330 switches[15] in our experiment. One is the under-detecting switch and the other two are the switches that send packets and capture packets. The experiment environment is as figure 4. The detection time is in table I.

Detection	Num. of Pkts	Exe. Time (msec)
Forwarding	1	108.75
Weighting	100	77.06
	1000	279.98
	10000	2258.23

TABLE I: Detection Time Cost.

In the forwarding detection algorithm, it takes only one packet to see if the packet is correctly forwarded. As for the weighting detection algorithm, it takes more packets to see the received packet ratio. From table I, we can see that the weighting detection algorithm does not cost more time than the forwarding detection algorithm. Actually, when there are only 100 packets, the weighting detection algorithm needs time less than the forwarding detection algorithm. The reason may be the action of the capture flow entry. In these two detection algorithms, the capture flow entries' actions are set to *Controller* and *Drop* respectively and undoubtedly, *Drop* is faster than send the packet to the controller. Of course, in the weighting detection algorithm, precise ratio can be derived from more packets and more packets imply more time. The trade-off can be determined by the manager.

B. Detection Probability Analysis

Since our detection mechanism is a sampling approach, it is not guaranteed to find compromised switches in one detection process. In this subsection, we evaluate the probability that some compromised nodes are detected. Suppose there are n OpenFlow switches and each has a flow entries. Let an attacker compromise m out of n switches and c out of a flow entries of each compromised switch are maliciously altered. We make each detection process randomly selects l switches and b flow entries of each selected switch. Let P_i be the probability that exact i compromised switches are detected and let E be the expected value of detected switches in one detection process. P_i and E can be calculated as follows:

$$P_i = \sum_{j=0}^{\min(m,l)-i} \left(\begin{array}{l} P_{(i+j) \text{ compromised switches are selected}} \times \\ P_{i \text{ compromised switches are detected}} \end{array} \right)$$
$$= \sum_{j=0}^{\min(m,l)-i} \frac{C_{l-(i+j)}^{n-m} C_{i+j}^{m}}{C_l^n} \cdot C_j^{i+j} \left(\frac{C_b^{a-c}}{C_b^a}\right)^j \left(1 - \frac{C_b^{a-c}}{C_b^a}\right)^i,$$

$$E = \sum_{i=0}^{\min(m,l)} i \cdot P_i.$$

Now we see how these parameters affect the expected detected switches. We discuss two cases, the densely compromised case and the sparsely compromised case. In the densely compromised case, we make 10% switches are compromised. We make $n = 1000$, $m = 100$ and $a = 10K$. We let our detection process randomly select 50 switches as the under-detecting switches. Here we see how the sampling flow entry number and the compromised flow entry number affect the performance of our detection mechanism. Since the network is densely compromised, we use the expected captured switch number as the evaluation score. Figure 5 is the evaluation result. Undoubtedly, by given a compromised switch, it is with higher probability to determine that the switch is compromised if the attacker spoofed more flow entries or the detection process samples more under-detecting entries. When the probability of a compromised switch being detected, the expected detected switch number is close to $l \cdot \frac{m}{n}$. If the attacker just cracks few flow entries, it takes more sampling entries to catch compromised switches.

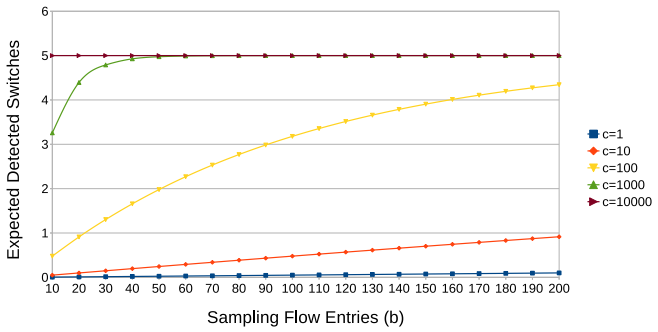


Fig. 5: Expected Detected Switch vs. Sampling Flow Entries in the densely compromised case in one detection process. There are total 1000 OpenFlow switches and 100 out of them are compromised. Each switch has 10K flow entries. The sampling switch number is set to 50.

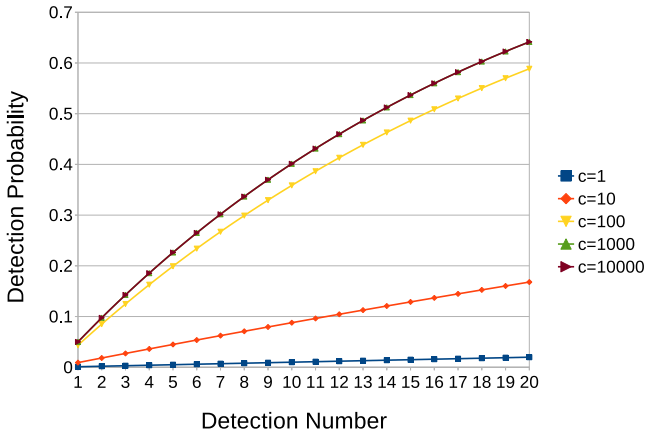


Fig. 6: Detection Probability vs. Detection Number in the sparsely compromised case. There is only one compromised switch out of 1000 switches. Each switch has 10K flow entries. The sampling switch number is set to 50.

Now we see the sparsely compromised case. We make only one switch that is compromised out of 1000 switches. We let our detection process randomly select 50 switches as the under-detecting switches and 100 entries from each switch. Here we see how the sampling flow entry number and the detection number affect the performance of our detection mechanism. We use the probability of capturing the compromised switch as the evaluation score. Figure 6 is the evaluation result. We can see with more detection processes, our detection mechanism can at last capture the compromised switch.

From the evaluation results, we can see the most difficult case for our detection mechanism is that a compromised switch just maliciously handles few network flows. To enhance the detection ratio in this case, we can increase the flow entry sampling rate. However, this will cause longer detection time.

V. Conclusions and Future Works

In this paper, we describe how compromised switches can spoil networks. We propose several attack models and design two detection algorithms for finding out compromised switches. Since our detection mechanism uses existing OpenFlow protocols without any modification, we implement our algorithms as OpenFlow APPs and these APPs can be executed while the network is operating. Therefore, our mechanism can be executed online and is practical to be applied to OpenFlow networks.

There are some future works on the compromised OpenFlow switch detection topic:

- **More Attack Models.** In this paper, we list three malicious actions as our first step. There may be more attack models. We will do further researches on discovering more attack behaviors and design corresponding detection algorithms. We will integrate future algorithms on our current OpenFlow platform and make our detection system more complete.
- **Pre-Detection Process.** As our evaluation results, more sampling flow entries can raise the detection probability. However, more sampling flow entries means that it takes longer time to run the detection process. Our next step is to enhance our detection mechanism to a hybrid detection system by adding a pre-detection process. The pre-detection process will be designed to rate all OpenFlow switches of being compromised. For example, a switch in a physically safe room may not easily be compromised. Another example is to dynamically rate all switches by continuous monitoring networks. After the rating process, we can raise the sampling flow entry number for those switches while other switches remain normal sampling flow entry number and run the detection process in this paper. By this hybrid idea, the detection mechanism can spend more time on suspicious switches.
- **Compromised Node Collaboration.** In our detection algorithms, we rely on all switches following on our detection commands to test an under-detecting switch. If two compromised switches collaborate, that is, one switch may help the other switch to pass the detection process, our detection algorithms will not work. To solve this problem, we may use switches multiple hops away from the under-detecting switch to decrease the probability of choosing compromised nodes. Moreover, by adding transmission hops, it may test several switches in one detection process.

To conclude, this paper is just a first step toward compromised OpenFlow detection. We will continue this work for further enhancement.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] "OpenFlow Switch Specification," Open Networking Foundation, Mar. 2014.

[3] "Opflex: An open policy protocol," 2014.

[4] "Ryu sdn framework." [Online]. Available: http://osrg.github.io/ryu/

[5] M. Shtern, R. Sandel, M. Litoiu, C. Bachalo, and V. Theodorou, "Towards mitigation of low and slow application ddos attacks," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 604–609.

[6] S. Lim, J. Ha, H. Kim, Y. Kim, and S. Yang, "A sdn-oriented ddos blocking scheme for botnet-based attacks," in *Ubiquitous and Future Networks (ICUFN), 2014 Sixth International Conf on*. IEEE, 2014, pp. 63–68.

[7] M. Vizváry and J. Vykopal, "Future of ddos attacks mitigation in software defined networks," in *Monitoring and Securing Virtualized Networks and Services*. Springer, 2014, pp. 123–127.

[8] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou, "Ddos attack protection in the era of cloud computing and software-defined networking," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. IEEE, 2014, pp. 624–629.

[9] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 161–180.

[10] K. Giotis, G. Androulidakis, and V. Maglaris, "Leveraging sdn for efficient anomaly detection and mitigation on legacy networks," in *Third European Workshop on Software Defined Networks*, 2014.

[11] J. Matias, J. Garay, A. Mendiola, N. Toledo, and E. Jacob, "Flownac: Flow-based network access control," in *Third European Workshop on Software Defined Networks*, 2014.

[12] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, "Let SDN be your eyes: Secure forensics in data center networks," in *Proceedings of the NDSS Workshop on Security of Emerging Network Technologies (SENT'14)*, Feb. 2014.

[13] D. Yu, A. W. Moore, C. Hall, and R. Anderson, "Authentication for resilience: The case of sdn." in *Security Protocols Workshop*, ser. Lecture Notes in Computer Science, B. Christianson, J. A. Malcolm, F. Stajano, J. Anderson, and J. Bonneau, Eds., vol. 8263. Springer, 2013, pp. 39–44.

[14] M. Antikainen, T. Aura, and M. Särelä, "Spook in your network: Attacking an SDN with a compromised openflow switch," in *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, 2014, pp. 229–244.

[15] "Centec v330 series switch." [Online]. Available: http://www.centecnetworks.com/en/SolutionList.asp?ID=42