

Inferring OpenFlow Rules by Active Probing in Software-defined Networks

Po-Ching Lin, Ping-Chung Li, Van Linh Nguyen

Department of Computer Science and Information Engineering

National Chung Cheng University

Chiayi, Taiwan 62102

Email: pclin@cs.ccu.edu.tw, x30346@gmail.com, nvlinh@cs.ccu.edu.tw

Abstract—Software-defined networking (SDN) separates the control plane from underlying devices, and allows it to control the data plane from a global view. While SDN brings conveniences to management, it also introduces new security threats. Knowing reactive rules, attackers can launch denial-of-service (DoS) attacks by sending numerous rule-matched packets which trigger `packet-in` packets to overburden the controller. In this work, we present a novel method “INferring SDN by Probing and Rule Extraction” (INSPIRE) to discover the flow rules in SDN from probing packets. We evaluate the delay time from probing packets, classify them into defined classes, and infer the rules. This method involves three relevant steps: *probing*, *clustering* and *rule inference*. First, forged packets with various header fields are sent to measure processing and propagation time in the path. Second, it classifies the packets into multiple classes by using k-means clustering based on packet delay time. Finally, the apriori algorithm will find common header fields in the classes to infer the rules. We show how INSPIRE is able to infer flow rules via simulation, and the accuracy of inference can be up to 98.41% with very low false-positive rates.

I. INTRODUCTION

Software-defined networking (SDN) separates the control logic from underlying devices and promotes innovations to be rapidly deployed in operational networks. Due to the layered structure of SDN, a controller configures underlying switches through the southbound interface, typically the OpenFlow protocol. Each switch has at least a flow table to store rules determined by the controller. The rules specify actions enforced on incoming and outgoing traffic.

Normally, attackers launch reconnaissance first before starting an attack, and so do they in SDN. For example, they can scan a network to gather forwarding policies. The more information they have, the more strategies they can use. Probing is required to know configured rules of a targeted firewall [1], [2]. Getting firewall rules successfully, they can pass malicious data through the firewall. Likewise, SDN faces a similar problem because the rules also reflect forwarding policies over network.

Nonetheless, OpenFlow rules are different from firewall rules in the following respects: (1) Firewalls usually support only IP addresses, port numbers and TCP flags, but OpenFlow supports over 40 fields (OpenFlow v1.5.1 [3]). Thus, the space of fields in OpenFlow rules is much larger than that in firewall rules. (2) The actions involve only “permit” and “deny” in a firewall, but an OpenFlow switch can forward

packets to others, adjust field values (e.g., TTL) of specific packets, and so on. (3) Administrators install the firewall rules proactively by default, while the OpenFlow rules are marked as *proactive* or *reactive* [3]. The proactive rules are also installed in advance, but the reactive rules will be installed or updated into OpenFlow switches only if a packet does not match any flow entries in a flow table and triggers an interaction between the OpenFlow switches and the controller by a `packet-in` packet. We summarize the differences between OpenFlow switch’s rules and firewall’s in Table I.

TABLE I
THE DIFFERENCES OF RULES IN AN OPENFLOW SWITCH AND A FIREWALL

	OpenFlow switch	Firewall
rule actions	forward, drop, relabel, etc.	permit or deny
number of fields	more	fewer
rule table	one or more	only one
rule installation	proactive or reactive	proactive

Recently, some studies have started to investigate SDN reconnaissance. Leng et al. [4] proposed a novel attack at flow table capacity and usage. Cui et al. [5] devised a method to fingerprint a specific OpenFlow rule by measuring round-trip-time (RTT) and packet-pair dispersion of the probing packets from an Internet host. However, both lack the overall perspective about the flow rules in the target network. In this work, we pretend to be adversaries and try to perform reconnaissance against OpenFlow rules. We send probing packets with crafted headers to the target network. After getting received packets and evaluating delay time, we use *k-means clustering* to classify packets into three major classes: *blocked*, *proactive* and *reactive*. The first means the probing packets are blocked somewhere, the second means they match only proactive rules, and the last means one or more reactive rules are matched in the probing path.

The packets in each class may have common header fields. Based on this, we infer the flow rules by using the *apriori algorithm* [6] to identify those header fields based on their distributions and compose the rules in each class accordingly. Attackers can exploit the inferred rules in the reactive class to launch DoS attacks by sending numerous forged packets that will trigger `packet-in` packets from OpenFlow switches to overburden the controller in a short time.

The remaining parts of the paper are organized as follows.

We briefly review reconnaissance techniques in firewalls and SDN in Section II. Section III presents the INSPIRE method, which includes the probing strategy, classification of sent packets and rule inference from the probing results. In Section IV, We illustrate the experimental settings and analyze retrieved results. Finally, we conclude this work and recommend future work in Section V.

II. RELATED WORK

In this section, we will review the background knowledge about how to probe firewall rules and then OpenFlow rules.

A. Firewall Policy Reconnaissance

Given the large space of header fields, scanning all the possible values by brute force is certainly inefficient. Strategies to increase this efficiency are reviewed below.

1) *Region Growing Method*: Li et al. [1] assume that firewall policies are “deny” by default. They attempt to find a packet which passes firewall filtering, i.e., a “permit” rule is found. From this rule, this method performs an exponential search in each dimension, e.g., IP address. This query will stop whenever a “deny” rule is found. Following the exponential search comes binary search to get the exact “permit” rule boundaries. Once the rule space is found, this method will recursively traverses the space other than the found “permit” rule space [7]. Finally, this method will stop after the time limitation or the maximum number of packets that the attacker desires to send. However, the error is over 12% in identifying the “permit” rules.

2) *Split-and-Merge Method*: This method was originally proposed for image segmentation. Samak et al. [1] showed that it can be applied to firewall reconnaissance. First, it will consider the entire rule region as R , which contains only “deny” rules. Then, it splits R into non-overlapping sub-regions (sub-rules) based on certain criterion $R_1, R_2, \dots, R_n$. After that, the prediction is performed in each sub-region to evaluate whether it contains “deny” rules or “permit” rules. If the prediction is a “permit” sub-region, this sub-region will be merged to other “permit” ones. This merging step continues spreading to the whole region. The error rate of identified rules is around 16%.

3) *Hybrid Method*: Ali et al. [8] proposed a hybrid method that combines both split-and-merge and region growing. This method has two steps. First, it splits the rule space for maximum exploration, and then region growing in each block is used to refine the rule boundaries. The maximum number of probing packets are user-defined. The authors proved that the hybrid method can take advantage of both prior methods, but the false alarms rate of detectors is unacceptably high, over 40% in some scenarios. Combination only works in some specific environments.

4) *Line Sweep Method*: Kim et al. [2] proposed the line sweep algorithm to probe firewall policies. Its features are fast detection to recognize vertical sweep attacks. It searches space in only $n - 1$ dimensions in n -dimensional space. This algorithm generates packets to cover the sweep line, which has

an angle of 45 degrees across the space. After finishing the first line, it will keep generating another line and packets to cover the line until all the space has been searched completely. Its results reflect that distinction between inbound and outbound rules is complicated to identify.

B. SDN Reconnaissance

1) *Fingerprinting SDN*: Adversaries can measure the forwarding rules of a network from the external due to different packet processing time on the OpenFlow switches. Cui et al. proposed a method to fingerprint a data center network (DCN) by RTT and packet-pair dispersion of probing packets [5]. Their architecture contains three hardware OpenFlow switches and one software OpenFlow switch, all connected to a controller to simulate a layered structure. The controller renews the flow rules on the switches periodically, and the probing packets sent from the outside trigger rule installation on some switches. Finally, they found a threshold to define if that controller-switch interaction happens. Their simulated results proved SDN is fingerprintable, but the authors did not mention how to infer the whole flow rules.

2) *An Inference Attack Model for Flow Tables*: Leng et al. [4] showed that limited flow table capacity can cause serious SDN vulnerability. The attacked target is flow table capacity and usage because most OpenFlow switches have limited flow table capacity. A DoS attack will drop network performance if the flow table is full. By observing packet response time, it is feasible to infer the processing states inside a switch. The authors proposed algorithms to infer flow table capacity and usage, but that work did not deal with flow rules reconnaissance.

III. METHOD OF PROBING SDN FLOW RULES

In this section, we introduce the main idea, the design, the workflow, and point out the limitations.

A. Threat Model

Although cloud services can be open to the public as long as users pay for it, the infrastructure and internal working are still private to service providers. However, it is a challenge for providers to prevent users from probing their networks to learn the cloud internals because a crafty user can probe in diverse ways that look legitimate. Several studies have demonstrated the possibility by various probing techniques [9]–[12]. Likewise, attackers can pay for leased machines inside a cloud like normal users, and attempt to learn the forwarding rules on the internal switches. We assume the internal switches are all OpenFlow capable. The assumption is reasonable due to the trend of using SDN for flexible configurations [13], [14]. We also assume attackers do not compromise any components in the cloud for probing.

B. Architecture

When probing OpenFlow rules, similar to probing firewalls rules, we need to forge packets with various match fields and observe how the packets are handled by the rules to infer what

the rules may be. Suppose that a probing packet with source IP address IP_A and TCP destination port number $Port_A$ is blocked. It is still unknown from only one packet whether some rule blocks IP_A or $Port_A$, so we need to try with other IP addresses and $Port_A$ to know whether $Port_A$ is blocked, or other TCP ports and IP address IP_A to know whether IP_A is blocked. Therefore, multiple TCP packets are necessary to infer the rules about TCP ports and IP addresses correctly. We probe with the fields in the TCP/IP headers at the same time to find the rules. Likewise, we also probe with the header fields of other layer-4 protocols like UDP and SCTP.

In this work, the following header fields will be forged and tested: source IP address and source/destination ports of TCP, UDP and SCTP. It is noted that probing with various destination IP addresses is costly because we need to deploy a number of hosts with those destination IP addresses to receive probing packets. Therefore, we set the probing packets all toward a receiver deployed outside the SDN. However, the INSPIRE method can still probe with various destination IP addresses if the resources to deploy a large number of external hosts are available.

An OpenFlow switch supports rich actions to manage flows, but not all the actions are observable from the external. For example, looking up another flow table is executed inside the switch, and cannot be discovered by external probing. Fortunately, we can observe the packet delay time to infer whether a packet triggers `packet-in` reaction between the controller and a switch. The delay time is the most obvious feature which allows us to infer whether a packet triggers controller-switch interaction. We will classify which rules are installed reactively with delay time values.

In summary, the INSPIRE method involves four parts:

- 1) It infers the way of rule installation by measuring delay time difference (Section III-B1).
- 2) It leverages the line sweep method to craft the headers of probing packets (Section III-B2).
- 3) After probing, it uses clustering to classify the probing packets and group them by blocked, proactive, or reactive classes (Section III-B3).
- 4) Finally, it extracts the rules from each class by using the apriori algorithm (Section III-B4).

1) Measurement of Rule Installation: We set up a pair of hosts to send and receive probing packets. The sender is a rented host in an SDN-based cloud, and the receiver is located outside the cloud. The sender sends probing packets with spoofed source IP addresses to the receiver. The range of spoofed IP addresses is usually within the same subnet where the sender stays, but a larger range is also acceptable. The spoofing allows to learn the rules about source IP addresses. Each probing packet contains a sending timestamp in the packet payload. The timestamp will remain intact during the packet traveling because OpenFlow switches do not override any packet payload.

The time difference is derived by subtracting the sending time in the packet payload from the packet receiving time. Thus, it is critical that the system time on the sender and the

receiver should be synchronized beforehand, e.g., by using the network time protocol (NTP). Furthermore, the receiver does not send any response back to the sender. There are two reasons for doing this:

- 1) Most of the probing packets are spoofed, so the response will not arrive the sender.
- 2) The sender does not have to wait for the response, so we do not need to define how long the sender should wait.

If a rule is proactive, the packet processing time will be shorter than that of a reactive rule because the switch just looks up the flow table without interacting with the controller. The time difference helps to infer not only whether a rule is stored in the flow table, but also to how the rule is installed as well.

Note that INSPIRE relies on forged packets. If spoofed packets are prohibited in a cloud environment [15]–[18], there is still another way to work around it. Ristenpart et al. [11] presented a co-residence vulnerability in Amazon EC2, and revealed the internal address space in EC2. It allows an adversary to associate IP addresses with various instance types. Xu et al. [12] demonstrated this threat still exists in 2015. With the characteristics of co-residence and the mapping of internal IP addresses, a certain range of source IP addresses can be acquired by launching different types of machines (at the cost of a higher fee to rent the hosts) or restarting an instance many times. Thus, an attacker will be able to acquire various (real) source IP addresses for probing in the INSPIRE method.

2) Selecting the Probing Points: To generate a series of probing packets, we use the line sweep method to determine the header fields of next probing packet. This method is fast to discover the rules in a large block and can evade port scanning detection because of the diagonal scanning direction [19]. We use this method because region growing uses both exponential and binary search, and small-ranged rules may be skipped between two probing points far away from each other. Moreover, line-sweep will generate more probing packets than split-and-merge to decrease outliers in the dataset, since network congestion may affect the delay time.

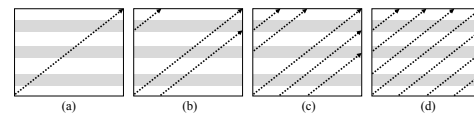


Fig. 1. Scanning direction in line sweep [19].

Instead of port or IP sweeping, the line sweep method selects various IP addresses and ports for probing packet headers. The source IP address and destination port are incremented sequentially and simultaneously to generate next probing packet. Figure 1 presents the scanning direction and orbit of line sweep. The scanning direction in INSPIRE fairly similar to that in Figure 1, but the x-axis is source IP addresses and the y-axis is destination port numbers in each sub-graph. Even though OpenFlow supports over 40 protocol fields, a packet cannot contain them all (e.g., TCP and UDP headers

are exclusive). We select only IP addresses and ports to forge because these fields are usually related to forwarding decision. The time complexity of INSPIRE, like that of the line sweep method, is $O(nm)$ in the worst case, where n is the number of source IP addresses and m is the number of ports.

3) *Clustering and Classification*: After the probing process finishes, INSPIRE will classify the probing packets. Both the sender and receiver are under our control, so it is trivial to know which probing packets are blocked. For those captured packets on the receiver, we classify them into either the proactive or the reactive class with *k-means clustering* based on the difference of the receiving time and the sending time for each packet. The clustering divides the captured packets into two classes: the proactive class and the reactive class. The latter can be further divided into n sub-classes if several switches (assuming n switches) are in the probing path. Each switch may trigger `packet-in` to the controller, so the total delay time will depend on how many switches will interact with the controller. In this work, we evaluate the accuracy of inference based only on the reactive class because knowing the rules in this class is sufficient for triggering `packet-in` to the controller. A detailed evaluation of accuracy based on the reactive sub-classes is left to future work.

4) *Inferring Flow Rules from Packets*: Finally, we use the *apriori* algorithm to extract the rules [6]. It counts the occurrence frequencies of the header fields and also looks for combinations of header fields with high occurrence frequencies as part of the rules. For example, if a rule specifies every packet with TCP port 22 should be sent to the controller in `packet-in`, the reactive class must have many packets with TCP port 22. The combinations of other fields and TCP port 22 are likely to have low occurrence frequency and will be finally excluded from the rules. The *apriori* algorithm runs the process until the occurrence frequencies of field combinations are all less than a pre-defined minimum support or no more fields can be combined.

C. Restrictions in This Work

Well-managed data centers have deployed systems to protect tenants against DDoS attack from the outside or the other tenants [9]–[12]. They also prevent tenants from sending spoofed packets. This protection may impose restrictions on this work. As discussed in Section III-B1, several techniques are feasible to probe with real source IP addresses, and the probing speed can be slowed down to evade DDoS detection. Nonetheless, the efficiency of probing process will be still affected.

IV. EVALUATION

In this section, we will evaluate the effectiveness of this work. We will describe the environment setup for evaluation first, and then evaluate the accuracy with various factors such as bandwidth, link delay and background traffic.

A. Environment Setup

We set up a simulated DCN with a layered structure for evaluation. The topology contains three layers of OpenFlow

switches. From the top layer to the bottom are core switches, aggregation switches and edge switches. A real-world DCN contains much more switches in each layer, but the probing path length in it is still the same as our simulated one (i.e., three hops). Table II summarizes the tools in our evaluation. The simulator is Mininet (mininet.org), which creates the simulated network with virtual OpenFlow switches. The controller is Ryu (osrg.github.io/ryu), a component-based SDN framework. Scapy is a powerful packet manipulation library to generate probing packets and background traffic.

TABLE II
SUMMARY OF EXPERIMENTAL TOOLS

Items	Tools
Operating System	Ubuntu 14.04 x86_64
Controller	Ryu 3.26
OpenFlow Switch	OpenvSwitch 2.4.0
Southbound API	OpenFlow 1.3
Network Emulator	Mininet 2.2.1
Packet Generator	Scapy 2.2.0

Figure 2 illustrates the simulated environment for evaluation. Seven OpenFlow switches (from switch 1 to switch 7) are in the simulated DCN. They will follow the instructions from the controller, and communicate with it when a packet triggers `packet-in`. We also assume eight hosts are deployed inside the DCN and two hosts are in the external. One of the internal hosts is the sender for probing, and the other internal hosts generate background traffic to simulate user traffic from numerous sources in the DCN with spoofed packets. Some packets in the background traffic may also trigger `packet-in` to the controller. The flow of probing packets is described in Section III-B. One external host is the receiver to receive the probing packets from the sender, and the other is the sink to receive the background traffic. The

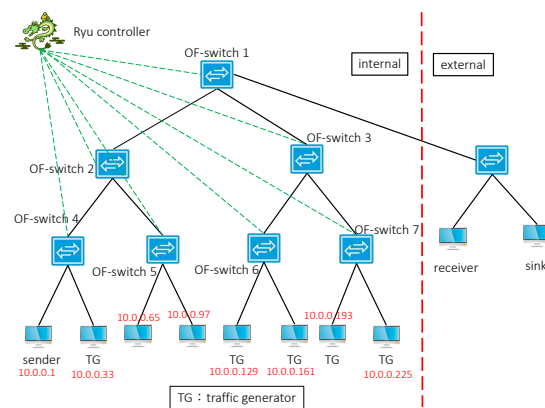


Fig. 2. The simulated DCN environment for evaluation.

OpenFlow rules in the experiment are presented in Table III. They specify what kind of packets should be sent to the controller or forwarded to the next hop (i.e., the proactive rules subsumed to “otherwise” in the last row). If a packet does not match any proactive/blocked rules on a switch, it will trigger `packet-in` to the controller, which will install a reactive rule

on the switch. Reactive rules are stored only in the controller in the beginning, waiting for a switch to trigger `packet-in` to acquire and install them. The last column indicates how many switches in the probing path have the proactive/blocked rules or will interact with the controller to install a reactive rule with the match fields and values. We will attempt to infer the rules and estimate the accuracy of this inference.

TABLE III
RULES TABLE

Match fields and values	Rule type	# of switches
TCP dst. port 200~220	Reactive	1
TCP dst. port 130~150	Reactive	2
TCP dst. port 20~40	Reactive	3
UDP dst. port 100~120	Reactive	3
SCTP 10.0.0.150~200 dst. port 200~254	Reactive	3
IP 10.0.0.64/26	Blocked	3
otherwise	Proactive	3

B. Packet Classification

The probing packets are classified by clustering into specific classes and sub-classes as defined in Section III-B3. For example, in a scenario with 50ms link delay, 1Gbps link bandwidth and one background traffic generator, the classification results are illustrated in Figure 3. Each dot represents a probing packet captured by the receiver. The colors of the dots represent the classes or sub-classes to which the dots belong. The blank rectangle region represents blocked packets, which are consistent with the blocked rules in Table III. The green, yellow and red dots are also roughly consistent with the reactive rules in Table III. According to this figure, we can infer how the flow rules are set and where the reactive rules are. The accuracy of inference will be evaluated in the next subsection. Some proactive rules may be classified as reactive ones incorrectly. The false-positive rates are almost within 3% and most are 0% in the following scenarios. Due to the space limitation, we do not show the false-positive rates of individual scenarios

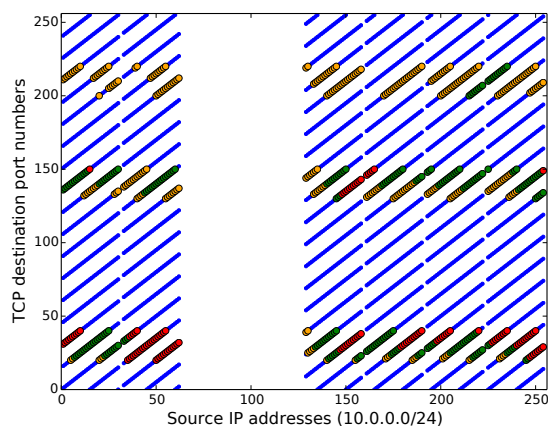


Fig. 3. Classification results of the probing packets (large yellow dot: a reactive rule on one switch; large green dot: reactive rules on two switches; large red dot: reactive rules on three switches; small blue dot: proactive rules).

C. Accuracy of Inference

In this subsection, we will present the probing result and evaluate the accuracy of inference in three aspects: link delay, bandwidth, and background traffic. To simplify the evaluation, we select TCP reactive rules as the target, and the sender only sends TCP probing packets. There are 63 TCP reactive rules for various destination ports in Table III. The accuracy is defined as the percentage of correctly inferred rules in the TCP reactive rules.

1) *Link Delay*: To observe how various link delays will affect the accuracy of INSPIRE, we set them to the following values: 20ms, 30ms, 40ms and 50ms. We assume that the delay of every link is the same in the layered structure. In this experiment, the bandwidth of the links is all 1Gbps, and we turn on only one traffic generator. The impact of multiple traffic generators on the accuracy will be discussed in Section IV-C3.

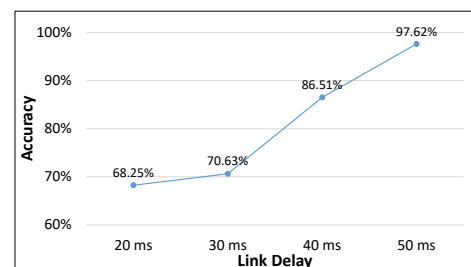


Fig. 4. Accuracy for various link delays.

From Figure 4, we see that the accuracy will be higher with longer link delay. If the link delay is 20ms, this accuracy is only 68.25%, but jumps to 97.62% if the delay time is 50ms. The observation can be explained as follows. If a packet triggers `packet-in`, the OpenFlow switches will spend additional time in acquiring reactive rules from the controller. Thus, longer link delay increases the entire interaction time between the control and data planes, and it is key to distinguish reactive rules and proactive rules.

2) *Bandwidth*: We also study the influence of link bandwidth on INSPIRE, as presented in Figure 5. We select three levels: 10Mbps, 100Mbps and 1Gbps in the experiment. The delay of every link is 50ms, and only two traffic generators are active. According to the figure, bandwidth has merely a minor effect on the accuracy. This effect results primarily from transmission delay. Large bandwidth means short transmission delay and vice versa. Thus, lower bandwidth will lead to longer delay in the controller-switch interaction for reactive rules and make the distinction from matching proactive rules more apparent. Compared with Figure 4, it also presents that longer link delay implies higher accuracy.

3) *Background Traffic*: Background traffic is also an important factor. We select 30ms for link delay and 1Gbps bandwidth in this scenario, with multiple levels of background traffic from one to five generators (see Figure 2). In the Scapy execution on the simulated hosts, the maximum background traffic per generator involves over 1400 packets in one minute. As we can see in Figure 6, it indicates that when the network

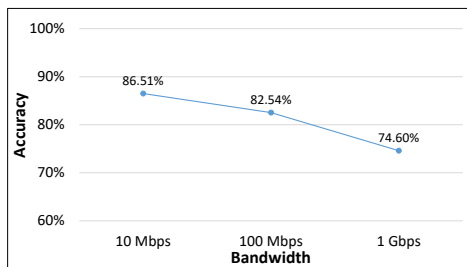


Fig. 5. Accuracy for various bandwidth.

is busy, the accuracy is higher, up to 98.41%. The reason is that when the background traffic increases steadily, packets that either match proactive rules or trigger reactive rules will take longer time to arrive the destination due to the increasing congestion in the path. The transmission of packet-in is no exception. Thus, it causes a similar effect of link delay.

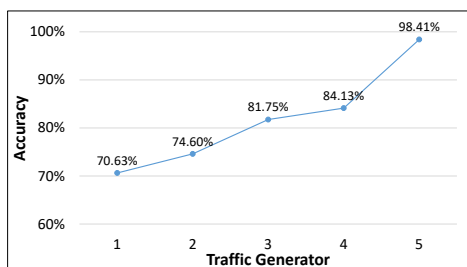


Fig. 6. Accuracy for various levels of background traffic.

V. CONCLUSION

In this work, we present INSPIRE to infer OpenFlow rules. Our main idea is based on the difference in delay time between the controller and the switches. For this purpose, we send probing packets to evaluate packet delay time and then classify them to infer flow rules. The evaluation demonstrates that our approach is suitable in a busy SDN. Compared with previous studies, INSPIRE can infer the flow rules in a DCN, not just a specific rule. Moreover, our method takes less time and needs fewer packets to accomplish the goal.

INSPIRE can be improved further in future work. First, clustering and classification of probing packets had better tell the number of switches in the probing path, and the accuracy of inferring rules in the sub-classes should be evaluated. Second, an efficient way is needed to determine the minimum support value for the apriori algorithm.

REFERENCES

- [1] T. Samak, A. El-Atawy, E. Al-Shaer and H. Li, "Firewall policy reconstruction by active probing: an attackers view," in Proc. 2nd Workshop Secure Netw. Protocols, Nov. 2006.
- [2] H. Kim and H. Ju, "Efficient method for inferring a firewall policy," 13th Asia-Pacific Network Operations and Management Symposium, 2011.
- [3] OpenFlow Switch Version 1.5.1 Specification, <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf>.
- [4] J. Leng, Y. Zhou, J. Zhang and C. Hu, "An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network," arXiv preprint arXiv:1504.03095, 2015.

- [5] H. Cui, G. O. Karame, F. Klaedtke and R. Bifulco, "Fingerprinting Software-defined Networks," arXiv preprint arXiv:1512.06585 (2015).
- [6] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," the 20th Intl. Conf. Very Large Data Bases (VLDB), 1994.
- [7] T. Samak, A. El-Atawy and E. Al-Shaer, "FireCracker: A framework for inferring firewall policy using smart probing," Proc. IEEE ICNP, Oct. 2007.
- [8] M. Q. Ali, E. Al-Shaer and T. Samak, "Firewall Policy Reconnaissance: Techniques and Analysis," IEEE Transactions on Information Forensics and Security, vol. 9, issue 2, pp. 296–308, Jan. 2014.
- [9] V. Varadarajan, Y. Zhang, T. Ristenpart and M. Swift, "A placement vulnerability study in multi-tenant public clouds," the 24th USENIX Security Symposium, Aug. 2015.
- [10] A. Khandelwal, N. Jain and S. Kamara, "Attacking Data Center Networks from the Inside," <http://research.microsoft.com/apps/pubs/default.aspx?id=249024>.
- [11] T. Ristenpart, E. Tromer, H. Shacham and S. Savage, "Hey, you, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," the 16th ACM Conference on Computer and Communications Security (CCS), Nov. 2009.
- [12] Z. Xu, H. Wang and Z. Wu, "A Measurement Study on Co-residence Threat inside the Cloud," the 24th USENIX Security Symposium, Aug. 2015.
- [13] H. Kim and N. Feamster, "Improving Network Management with Software Defined Networking," IEEE Communications Magazine, vol. 51, issue 2, pp. 114–119, Feb. 2013.
- [14] G. Wang, T. S. E. Ng and A. Shaikh, "Programming Your Network at Run-time for Big Data Applications," the First Workshop on Hot Topics in Software Defined Networks (HotSDN), Aug. 2012.
- [15] Amazon Web Services: Overview of Security Processes, <https://d0.awsstatic.com/whitepapers/aws-security-whitepaper.pdf>.
- [16] Microsoft Azure Network Security, http://download.microsoft.com/download/C/A/3/CA3FC5C0-ECE0-4F87-BF4B-D74064A00846/AzureNetworkSecurity_v3_Feb2015.pdf.
- [17] Rackspace Private Cloud Security White Paper, <https://support.rackspace.com/white-paper/rackspace-private-cloud-security-white-paper>.
- [18] Security at Linode, <https://www.linode.com/security>.
- [19] H. Kim, W. Pak and H. Ju, "Correlation Analysis between Inference Accuracy and Inference Parameters for Stateless Firewall Policy," 15th Asia-Pacific Network Operations and Management Symposium (APNOMS), Sept. 2013.



Po-Ching Lin received the Ph.D. degree in computer science from National Chiao Tung University, Hsinchu, Taiwan, in 2008. He joined the faculty of the Department of Computer and Information Science, National Chung Cheng University (CCU), Chiayi, Taiwan, in August 2009. He is currently an associate professor. His research interests include network security, network traffic analysis, and performance evaluation of network systems.



Ping-Chung Li received his master's degree in Computer Science and Information Engineering from National Chung Cheng University in 2016. His research interests include SDN security.



Van Linh Nguyen received the B.E and M.S.c degree in Data Communication and Networking from University of Information and Communication Technology and University of Engineering and Technology, Vietnam National University, respectively. He is currently working towards Ph.D. degree in Computer Science and Information Engineering at National Chung Cheng University, Taiwan. His research interests are in the areas of network security, Internet of Things and Mobile Edge Computing. He is a student member of the IEEE.