Fingerprinting OpenFlow Controllers: The First Step to Attack an SDN Control Plane

Abdelhadi Azzouni, Othmen Braham, Thi-Mai-Trang Nguyen, Guy Pujolle, Raouf Boutaba

▶ **To cite this version:**

Abdelhadi Azzouni, Othmen Braham, Thi-Mai-Trang Nguyen, Guy Pujolle, Raouf Boutaba. Fingerprinting OpenFlow Controllers: The First Step to Attack an SDN Control Plane. 59th annual IEEE Global Communications Conference (GLOBECOM 2016), Dec 2016, Washington DC, United States. pp.1-6, 10.1109/GLOCOM.2016.7841843 . hal-01538464

HAL Id: hal-01538464
https://hal.sorbonne-universite.fr/hal-01538464

Submitted on 13 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fingerprinting OpenFlow controllers: The first step to attack an SDN control plane

Abdelhadi Azzouni[1], Othmen Braham[2], Nguyen Thi Mai Trang[1], Guy Pujolle[1], and Raouf Boutaba[3]

[1]LIP6 / UPMC; Paris, France {abdelhadi.azzouni,thi-mai-trang.nguyen,guy.pujolle}@lip6.fr
[2]Virtuor; Paris, France othmen.braham@virtuor.fr
[3]University of Waterloo; Waterloo, ON, Canada rboutaba@uwaterloo.ca

***Abstract*—Software-Defined Networking (SDN) controllers are considered as Network Operating Systems (NOSs) and often viewed as a single point of failure. Detecting which SDN controller is managing a target network is a big step for an attacker to launch specific/effective attacks against it. In this paper, we demonstrate the feasibility of fingerpirinting SDN controllers. We propose techniques allowing an attacker placed in the data plane, which is supposed to be physically separate from the control plane, to detect which controller is managing the network. To the best of our knowledge, this is the first work on fingerprinting SDN controllers, with as primary goal to emphasize the necessity to highly secure the controller. We focus on OpenFlow-based SDN networks since OpenFlow is currently the most deployed SDN technology by hardware and software vendors.**

keywords - Software-Defined Networking, OpenFlow, Control Plane, security.

I. Introduction and Motivation

Software-Defined Networking (SDN) is an emerging architecture that is dynamic, agile, centrally managed and programmable, making it ideal for the evolving nature of today's applications. SDN separates the network control plane from the data plane (Fig. 1), enabling more flexibility in managing and programming the network. The centralized control provided by SDN is expected to facilitate the deployment and hardening of network security [1], [2]. However, SDN controllers can be subject to new threats compared to conventional network architectures. For example, an attacker can change the whole underpinning of the network traffic behavior by modifying the controller. The Open Networking Foundation (ONF) identifies a number of SDN security issues that the community must address [3]:

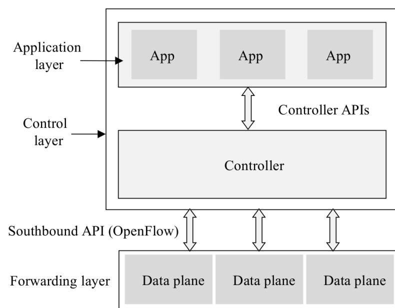

Fig. 1. SDN architecture

- The centralized controller emerges as a potential single point of attack that must be protected.
- The southbound interface between the controller and underlying networking devices (OpenFlow) is vulnerable to threats that could degrade the availability, performance, and integrity of the network. Using TLS or UDP/DTLS is recommended to secure the OpenFlow channel.
- The underlying network must be capable of enduring occasional periods where the SDN controller is unavailable.

In the most common schemes of attacking a remote system, the first step is to determine the set of possible attacks by collecting information about the target. In this paper, we demonstrate some techniques that allow an attacker to fingerprint the OpenFlow controller of the network. Once the attacker knows which controller is used, he/she can launch tailored attacks exploiting its known vulnerabilities. We study the common case where the attacker is placed in the underlying network managed by the target SDN controller, and does not have access to either the controller or the control channel.

This work aims to demonstrate the feasibility of fingerpirinting OpenFlow controllers, with the ultimate goal of building a Penetration Testing framework that can be used by network administrators to test their SDN networks. Many frameworks have been created for the same purpose in traditional networks, NMAP [4], for instance, is a widely used scanner that can fingerprint remote systems among other capabilities. OWASP Zed Attack Proxy Project (ZAP) [5] is another security testing framework that includes fingerprinting remote web servers and web applications. As in NMAP and ZAP, our proposed techniques are not to be used separately, that is, one may get non-accurate results when only using the first Timing-Analysis based technique (section IV-A1) for example, but the combination of all proposed techniques, generally gives accurate results. The contributions of this paper are as follows:

- We demonstrate the feasibility of fingerpirinting attack on OpenFlow controllers by designing, implementing and testing several fingerpirinting techniques and
- We highlight the need for building a Penetration Testing framework for SDN networks.

This paper is organized as follows. Section II provides OpenFlow background information. Related works are discussed in section III. Our proposed fingerpirinting techniques

are presented in section IV, our experimental testbed is described in section V and the results are given in section VI. Section VII concludes the paper and discusses some future directions.

II. Background Information

OpenFlow is the first standard communication interface defined between the control and forwarding layers of an SDN architecture [7]. Contrary to traditional routers, where the fast packet forwarding (data path) and the high level routing decisions (control path) occur on the same device. OpenFlow separates these two functions: An OpenFlow switch consists of one or more flow tables, which performs packet lookups and forwarding, and interfaces to external controllers. The controller controls the switch by adding, updating, and deleting flow entries via OpenFlow messages, proactively or reactively (in response to arrival of new flows) .

An OpenFlow message is either a switch-to-controller or a controller-to-switch message. OpenFlow messages are detailed in [8] of which the most important ones are:

- Hello messages: exchanged between the controller and the switch when the connection is first established.
- Echo request/reply messages: used to exchange information about latency, bandwidth and liveness.
- Packet-In messages: used by the switch to send a packet to the controller when it has no flow-table matching the packet.
- Packet-Out messages: used by the controller to inject packets into the data plane of a particular switch.
- Flow-mod messages: used by the controller to modify the state of an OpenFlow switch.
- Stats request messages: used by the controller to request information about individual flows.

III. Related Work

S. Shi and G. Gun developed SDN Scanner [9] which exploits the network header field change. If a client sends packets to an SDN network, this client will observe different response times, because the flow setup time can be added in the case of non-matching flow (i.e., there is no corresponding flow rule in the data plane: response time $T1$) compared to the case when the corresponding flow rule exists (response time $T2$). SDN Scanner collects the response times then uses statistical tests to compare them. Thus, if an attacker can clearly differentiate $T1$ from $T2$ then he/she can detect the SDN network (the presence of an SDN controller). The evaluations conducted in the paper showed that SDN Scanner can fingerprint 24 networks out of 28 (i.e., a fingerprinting rate of 85.7%). However, SDN Scanner does not detect the controller type. In addition, collecting accurate values of $T1$ and $T2$ is extremely hard in real-world WANs because of the many variables that affect the response time. As such this method may not be efficient in WANs. [10] leverages information from the RTT and packet-pair dispersion to fingerprint controller-switch interactions (i.e. whether an interaction between the controller and the switches has been triggered by a given packet) in a remote SDN network.

L. Junyuan et. al [11] propose techniques to infer key network parameters like flow table capacity and flow table usage. For example, when the flow table is full, extra interactions between controller and switch are needed to remove some of the existing flow entries to make room for new ones, which may result in a performance decrease of the network. An attacker can take advantage of the perceived performance change to launch more effective attacks. More specifically, knowing the flow table size and usage, the attacker can estimate with high accuracy how many packets he/she needs to generate per second to flood the flow table and the required time to fill it up. hence, he/she could choose and correctly configure their attacking tools. Contrary to [11], our methods aim to infer control plane parameters to fingerprint controllers, which is more critical and of higher impact.

IV. Fingerprinting OpenFlow Controllers

The main approach developed in this paper is to combine several techniques to fingerprint an SDN controller from its underlying data forwarding plane. Although our proposed techniques can be used separately, the accuracy of the results is much higher when combining them. Also, using only one technique may not give any result in some situations. In other words, each method has its success probability, and combining several techniques intuitively increases the probability of identifying the type of SDN controller used.

The following subsections present our techniques categorized into two classes: Timing-Analysis based techniques and Packet-Analysis based techniques.

A. Timing-Analysis based techniques

These techniques are based on time measurement to infer some indicative parameters of the controller.

1) Timeout Values Inference: Each flow entry has an idle_timeout and a hard_timeout field values associated with it. They indicate respectively the time in seconds after which the entry will be removed from the switch if no packet matches it, and the time after which to remove the entry anyway. These timeout values can be set and modified by application developers or network administrators. But, in most cases when the network or parts of the network only need a basic flow forwarding without additional traffic engineering logic, the network admins tend to use the forwarding applications that come with the controllers (typically L2-Switches) and the probability that they change these applications' parameters is fairly low. Note that in recent controllers, those forwarding elements even include some advanced features [12].

The idea is to infer flow-entry timeout values and compare them to known timeout values of different controllers (timeout database). The timeout database is constructed as follows: for open source controllers, default timeout values can be gathered from their code source or configuration files. For proprietary controllers, the default timeout values can easily be figured out by simply using the controller and directly measuring the

values. This method can be fairly accurate because of the low probability for default values to be modified by administrators.

To measure timeout values from an end-host in the underlying network, we propose the two following algorithms (algorithm 1 and algorithm 2). These algorithms consider network disruptions that may affect communication channels between end-hosts and the switch, and between the switch and the controller. Both algorithms require the ability to connect to another end-host in the same data plane (a pingable end-host). Algorithm 1 measures idle_timeout in two steps: first, it calculates RTT_avg (average Round-Trip Time using ping) in case when corresponding flow entries are installed in the switch. Measurements may be made for a configurable duration and/or number of probing packets n. Second, it measures RTT every $wait$ seconds. $wait$ value will be incremented by $step$ seconds until a significant difference between measured RTT and calculated RTT_avg is encountered. This difference means that the flow entry expired and the switch needed to call the controller asking how to handle the new ping. Final value of $wait$ matches the flow-entry $idle_timeout$ value. A more accurate version of the algorithm is conceivable by using a binary search around the final $wait$ value, but by using $step$ of $5ms$, the algorithm remains very accurate even without binary search.

Note that in some controllers, the default $idle_timeout$ value is set to 0 which means infinite, so the flow entry will never be removed. We found this in the Ryu controller and $Hydrogen$, an old version of $OpenDaylight$ [13]. In this case, after a number of iterations, the algorithm will decide that the $idle_timeout$ value is infinite and the controller may be Ryu or $Hydrogen$ version of $OpenDaylight$. The search space has been limited to two controllers in this case, but we need to apply more techniques to decide which one of them.

Algorithm 1 idle_timeout measurement

```
1:  Send first ping to install flow entry;
2:  Send n pings and calculate the average ping time RTT_avg;
3:  Wait wait seconds;
4:  Send one ping and calculate ping time T_ping
5:  if T_ping ≈ RTT_avg then //the flow entry still exists
6:      wait ← wait + step;
7:      Go to 3;
8:  else//idle_timeout expired and the flow entry removed
9:      idle_timeout = wait
10: end if
```

To measure $hard_timeout$ value, we first calculate the average of RTT time (RTT_avg) and $idle_timeout$ values as in algorithm 1. Second, we send one ping to install the flow entry in the switch. Then, we send a ping every $wait$ seconds such as $wait$ value is less then $idle_timeout$. As long as the RTT value is close to the average (RTT_avg), we continue to add $wait$ seconds to the $hard_timeout$ value initialized to zero. We stop when we find a RTT value which is significantly greater than (RTT_avg).

Algorithm 2 hard_timeout calculation

```
1:  hard_timeout ← 0 seconds;
2:  Calculate RTT_avg as in algorithm 1;
3:  Calculate idle_timeout as in algorithm 1;
4:  Send one ping to make the controller install flow entry;
5:  Wait wait seconds, wait must be less then idle_timeout;
6:  Send one ping and calculate ping time T_ping;
7:  if T_ping ≈ RTT_avg then //the flow entry still exists
8:      hard_timeout ← hard_timeout + wait
9:      Go to 5;
10: else//hard_timeout expired and the flow entry removed
11:     print hard_timeout
12: end if
```

The attacker then compares the measured values ($idle_timeout, hard_timeout$) to known timeout values of controllers to guess which controller is used.

2) Processing-Time Inference: Each SDN controller is programmed differently using different tools, libraries and frameworks, so that each controller has its own execution speed. In other words, when receiving packets from the data plane, each controller takes a different time to process those packets and reply back to the data plane. The idea of this technique is to use estimated packet-processing time to determine the controller. As we mentioned before, authors of [9] used timing to determine if a remote network is an SDN network based on the difference of RTT in two cases: presence and absence of flow entries. As it has been mentioned by the authors, it is very difficult to measure with high accuracy the RTT to a remote network in a WAN because of many potential sources of disruption that may result in random variations of RTT values. In our technique, these disruption sources are minimal since the attacker is placed in the data plane of the target controller. And unlike [9], our method uses some key parameters inferred from the network to estimate processing time with higher precision.

The main idea in our approach is to measure the response time of the target controller and compare it to the processing-time database created beforehand. The processing-time database is a table that associates each controller to its processing time. Like in the previous technique (timeout values inference), we need a pingable destination end-host in the same data plane (the best scenario is that the attacker controls the destination end-host as well to be sure that its processing time does not affect the measurements). To create the processing-time database, we use a simplified

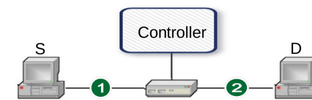

Fig. 2. Simplified architecture to measure controllers' processing time

architecture (Fig. 2) where the propagation times (1) and (2) are minimal. we first measure $idle_timeout$ and RTT_avg values as in algorithm 1. Then, we send n (100 for example) pings separated by $period$ seconds between every two

pings, with $period$ greater then $idle_timeout$. Every ping will cause the switch to send a Packet-In to the controller (by receiving the Packet-In message, the controller processes it to extract field values and installs the corresponding flow rule into the switch). Finally we calculate the average ping time T_{pavg} of the n pings and we record $T_{pavg} - RTT_avg$ value in a table. This is the processing time of the current controller. We repeat this process with all controllers and we create the processing-time database by inserting tuples $(controller, processing_time(T_p))$ (algorithm 3).

Algorithm 3 Building the processing-time database

1: Calculate RTT_avg as in algorithm 1;
2: Calculate $idle_timeout$ as in algorithm 1;
3: **for** $i \leftarrow 1..n$ **do**
4: Wait $period$ seconds, $period$ must be greater than $idle_timeout$;
5: Send a ping and save ping time;
6: **end for**
7: Calculate the average of saved ping time values T_{pavg} and calculate controller processing time $T_p = T_{pavg} - RTT_avg$;
8: Insert $(controller, T_p)$ in the processing-time database;

Note that, as propagation times (1) and (2) (Fig. 2) are minimal, measured RTT_avg is accurate and hence T_p values are accurate.

Algorithm 4 Fingerprinting $controller$

1: Calculate RTT_avg as in algorithm 1;
2: Calculate $idle_timeout$ as in algorithm 1;
3: **for** $i \leftarrow 1..m$ **do** //m = 20 for example
4: Wait $period$ seconds, $period$ must be greater then $idle_timeout$;
5: Send a ping and save ping time;
6: **end for**
7: Calculate the average of saved ping-time values RTT' and compare $RTT' - RTT_avg$ to the processing-time entries;

Now that we have the processing-time database, to fingerprint the target controller that manages the real SDN network we are connected in, we first measure RTT_avg to a destination, then we ping the same destination with a spoofed IP address to ensure that no corresponding flow entry exists in the switch and we compare the value $RTT - RTT_avg$ to the processing-time database entries. For accuracy, we do not rely on a single ping, disruptions can happen during the ping affecting the response time. Instead, we send many (20 for example) pings with $period$ seconds between every two pings (such as $period$ value is greater than $idle_timeout$), we calculate the average of these ping times RTT' and finally compare the value $RTT' - RTT_avg$ to the processing-time database entries (algorithm 4).

In addition to the probability that the network admin somehow modifies the execution time of the controller, which we argue is very low, there is a fair chance that during the scan, the controller is overloaded resolving requests and installing rules, which may significantly change the response time. In this case, if the attacker has further knowledge about the network state then he/she can surpass this problem. For example, he/she can avoid peak hours, and only scan the controller when the network is in its normal state.

B. Packet-Analysis based techniques

1) LLDP message analysis: This is a passive method which consists of identifying the controller by sniffing and analyzing OpenFlow Discovery Protocol ($OFDP$) packets sent over the data plane.

SDN is based on maintaining a global network view at the level of the controller. To obtain the global network topology, discovery modules of the controllers use $OFDP$ to collect updated information from different elements of the network including end hosts. $OFDP$ leverages the packet format of Link Layer Discovery Protocol ($LLDP$) with subtle modifications to perform topology discovery in an OpenFlow network.

Unlike ordinary $LLDP$ enabled switches, an OpenFlow switch needs the controller to send and process $OFDP$ messages and cannot do this by itself. The following is a simple scenario of the topology discovery process using $OFDP$. First, the SDN controller creates an individual LLDP packet for each port on each switch. Then, the controller sends these packets to the switches via Packet-Out messages that include instructions to send them out on the corresponding ports. In each switch, all received $LLDP$ packets will be forwarded to neighbours. When a switch receives a new $LLDP$ packet from another switch, it forwards it to the controller via a Packet-In message. At the end of the process, the controller will get information about all the data-plane connections. The entire discovery process is repeated periodically with the time periods varying from one controller to another, which is can be leveraged to identify which controller is managing the network. Also, the content of the $LLDP$ packets differs from one controller to another, which can be used accurately identify the controller. Table III in section VI-C shows $LLDP$ packets sent by different controllers.

2) ARP response analysis: This technique can only be used to determine if the controller is the Hydrogen version of OpenDaylight and cannot be generalized to other types of controllers. It builds on the observation of how the controller reacts to unknown Address Resolution Protocol (ARP) requests in the data plane. The attacker sends an unknown ARP request, which means that the destination IP address is not assigned to any host in the network. As the destination IP is not present in the network, the switch, in addition to broadcasting the request, sends it to the SDN controller via a Packet-In message asking how to handle it. The OpenFlow specifications indicate that the controller responds to the switch by a Packet-Out and/or a flow-mod message explaining how to handle the

request. The controller's response message differs from one controller to another, but the only controller whose behavior can be captured from an end-host is Hydrogen. Hydrogen version of OpenDaylight instructs the switch to broadcast the request once again which duplicates it in the broadcast domain. This duplicated ARP request, with one of the switch's Media Access Control (MAC) addresses as source address, indicates that Hydrogen is used.

As we mentioned in the introduction, the techniques we presented in this section are not to be used in an exclusive manner. Each technique is able to identify the controller with a certain probability that we did not compute analytically in this paper. The user can use a subset or all the techniques executing them one by one, or better combine them in some optimal order. The selection of the optimal combination of techniques is an interesting research question that we leave for future work.

V. Experiment Environment and Methodology

As shown in Figure 3, our experiment environment consists of four physical machines (only three are shown in Fig. 3) carrying 4 virtual machines each and connected via OpenFlow virtual bridges (Openvswitch) forming a small-size data-center where VMs generate random traffic (ping and iperf) to random destinations. Note that, since we are not exploiting any weaknesses in the switch, it does not make any difference using a virtual switch or a physical one in this context, we only need a switch that correctly implements OpenFlow specifications. Note also that we did not add hops (transit switches) between bridges and the controller because even in real-world networks, a very small number (0, 1 or 2) of transit switches is enough to build a fairly large Local Area SDN network, like a data-center or a campus network. Such a small number of hops does not affect timing measurements in previous algorithms, The attacker is on the red (or black) VM connected to $br0$, and can ping the orange (or dark grey) VM connected to $br2$ (it could be any other VM in the network).

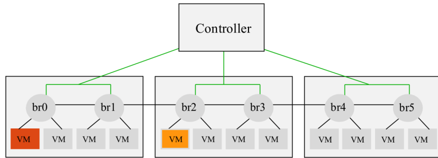

Fig. 3. Test environment

We have performed our experiments on five open source still maintained, OpenFlow controllers among the most widely used: OpenDaylight [13], POX [14], Beacon [15], Floodlight[17], and Ryu [18].

VI. Results

A. Timeout Values Inference technique

To evaluate the Timeout Values Inference method, we ran algorithms 1 and 2 ten times on each controller from the set of our target controllers. Default timeout values are given in table I. Our algorithms did 2 errors in 50 measurements in both $idle_timeout$ and $hard_timeout$ and that is because algorithm 1 is used in algorithm 2.

Controller	idle_timeout (s)	hard_timeout (s)
OpenDaylight	0	0
Floodlight	5	0
POX	10	30
Ryu	0	0
Beacon	5	0

TABLE I
Default Timeout Values

We also evaluated algorithms 1 and 2 separately by manually setting different values for $idle_timeout$ and $hard_timeout$ in POX source code, and running the algorithms from the attacker virtual machine (red VM in Fig. 3) to infer these values. We set the values $5, 10, 15, ..30ms$ for $idle_timeout$ and the values $10, 20, ..60ms$ for $hard_timeout$ respectively. For each algorithm, we repeated the execution 10 times on each value. $idle_timeout$ calculation algorithm has an error rate of 0.03% (2 errors in 60 measurements) with a relative error of less than $1s$. The $hard_timeout$ calculation algorithm has an error rate of 0% (no error) on 60 measurements.

B. Processing-Time Inference technique

First, we have built the processing-time database (table II) of our set of target controllers by running algorithm 3 ($n = 100$) on a simplified testbed as described in Fig. 2.

Controller	T_p (ms)	T_p adjusted (ms)
OpenDaylight	1.004	0.177
Floodlight	3.454	2.627
POX	34.266	33.439
Ryu	5.216	4.389
Beacon	3.197	2.370

TABLE II
Processing-time database (T_p: processing time).

Then, to evaluate this technique in our experimental environment (Fig. 3) we have run algorithm 4 ten times: measured T_p in Fig. 4 is the average value of the different executions. To get more precise comparisons, we calculate T_p adjusted: adjusted processing time = processing time - the average of RTT time in case the flow rule exists (RTT_{avg}).

For controllers Floodlight and Beacon which have very similar values of T_p, the use of only this technique, is not sufficient as it cannot decide between them.

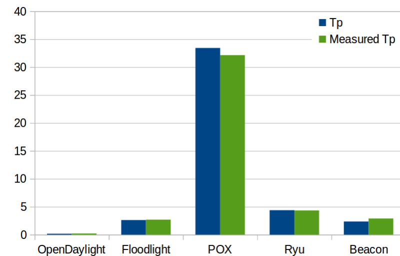

Fig. 4. Measured processing times compared to average processing times

Controller	$OFDP$ interval (s)	Remarks
OpenDaylight (Lithium & Helium)	5	LLDP packets include System Name field with value = "openflow" and no System Description field
OpenDaylight (Hydrogen)	300	LLDP packets include System Name field with value = "OF—[MAC address of the OF switch]" and no System Description field
Floodlight	15	Each LLDP packet is followed by an 0x8942 Ethernet packet sent in broadcast. This makes it easy to distinguish between Floodlight and Beacon
POX	variable (≈ 5)	LLDP packets include System Description field with value = "dpid:[MAC address of the OF switch]"
Ryu	1	Note that the Topology discovery module is still not stable and not included in the controller core.
Beacon	15	LLDP packets include two "unknown" fields and no System Name or Description feild

TABLE III
RESULTS OF $LLDP$ MESSAGE ANALYSIS

C. $LLDP$ message analysis technique

Figure 5 compares $LLDP$-packet reception intervals for different controllers. Table III shows the difference between controllers' $LLDP$ packets. By receiving the $LLDP$ packet, the attacker compares the different values against Fig. 5 and table III to identify the controller. Similar to technique IV-A1, for proprietary controllers, the way to gather $LLDP$ information is to simply use the controllers, analyze its $LLDP$ packets, then use this information to fingerprint target controllers.

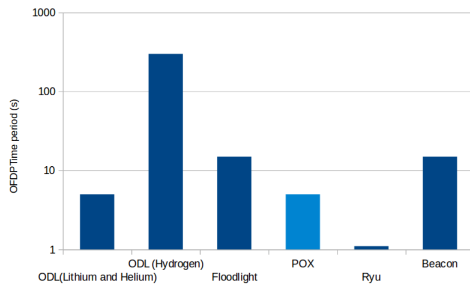

Fig. 5. Controllers' LLDP-emission-interval comparison

VII. CONCLUSION

In this work, we demonstrated the feasibility of fingerprinting attacks on OpenFlow controllers from the data plane by designing, implementing and testing practical techniques to identify the controller without access to the control plane. This is a critical step for a number of attack models since it provides the attacker with sufficient information about the controller to carry out more tailored attacks. Knowing the vulnerabilities of the target controller or one of its components, the attacker can indeed use known or design new attacks to take down the controller. In the future, we plan to expand the scope of this work by fingerprinting a larger set of controllers and by designing more techniques for fingerprinting controllers. We also plan to investigate formal methods for the evaluation of fingerprinting techniques and how they can be possibly combined to increase success rate. Finally, we plan to explore what countermeasures must be deployed to harden the security of SDN networks against controller fingerprinting and subsequent attacks.

REFERENCES

[1] Ahmad, Ijaz, et al. "Security in software defined networks: a survey." Communications Surveys & Tutorials, IEEE 17.4 (2015): 2317-2346.
[2] Scott-Hayward, Sandra, Sriram Natarajan, and Sakir Sezer. "A survey of security in software defined networks." (2015).
[3] Open Networking Foundation. "SDN Security Considerations in the Data Center", Version 1.4.0 (Wire Protocol 0x05). October 14, 2013
[4] NMAP. https://nmap.org/
[5] OWASP Zed Attack Proxy Project. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[6] Open Networking Foundation. "Software-Defined Networking". https://www.opennetworking.org/sdn-resources/sdn-definition.
[7] Open Networking Foundation. "OpenFlow". https://www.opennetworking.org/sdn-resources/openflow.
[8] Open Networking Foundation. "OpenFlow Switch Specification", Version 1.5.0 (Wire Protocol 0x06). December 19, 2014.
[9] Shin, Seungwon, and Guofei Gu. "Attacking software-defined networks: A first feasibility study." Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013.
[10] Bifulco, Roberto, et al. "Fingerprinting software-defined networks." 2015 IEEE 23rd International Conference on Network Protocols (ICNP). IEEE, 2015.
[11] Leng, Junyuan, et al. "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).
[12] Project Floodlight. https://floodlight.atlassian.net/wiki/display/floodlight-controller/Supported+Topologies
[13] Linux Foundation. "OpenDaylight". https://www.opendaylight.org/.
[14] Gude, Natasha, et al. NOX: towards an operating system for networks. ACM SIGCOMM Computer Communication Review 38.3 (2008): 105-110.
[15] Erickson, David. "The beacon openflow controller." In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, pp. 13-18. ACM, 2013.
[16] What is Beacon?. https://openflow.stanford.edu/display/Beacon/Home/
[17] Floodlight. http://Floodlight.openflowhub.org/
[18] Ryu. http://osrg.github.com/ryu/