

Timing SDN Control Planes to Infer Network Configurations

John Sonchack
University of Pennsylvania
jsonch@cis.upenn.edu

Adam J. Aviv
United States Naval Academy
aviv@usna.edu

Eric Keller
University of Colorado,
Boulder
eric.keller@colorado.edu

ABSTRACT

In this paper, we study information leakage by control planes of Software Defined Networks. We find that the response time of an OpenFlow control plane depends on its workload, and we develop an inference attack that an adversary with control of a single host could use to learn about network configurations without needing to compromise any network infrastructure (*i.e.* switches or controller servers). We also demonstrate that our inference attack works on real OpenFlow hardware. To our knowledge, no previous work has evaluated OpenFlow inference attacks outside of simulation.

1. INTRODUCTION

Software-defined networking (SDN) promises to transform the way networking is done by opening up the interfaces to network elements in a programmable way, and organizations such as Facebook, Google, and the NSA have already deployed large scale SDN networks. A key tenant of SDN is the separation of the control and data planes in a network. The *data plane* forwards packets across the network at high speeds by matching them against simple forwarding rules, while the *control plane* installs rules into the data plane and performs more advanced packet processing as needed (*e.g.* when a switch cannot determine how to forward a packet).

Packet processing in the control plane is orders of magnitude slower than packet processing in the data plane. These timing differences can leak information about a network. Previous work has shown how an adversary can leverage these timing differences to infer whether a network runs OpenFlow [4], how large its switches' forwarding tables are [3], and whether it contains links that aggregate flows [2].

In this paper, we study a more sophisticated inference attack that an adversary can use to learn more about a network. Our core observation is that *control plane load affects how long the control plane takes to process a packet*. An adversary can time the control plane while injecting packets into an SDN to determine whether the injected packets are invoking the control plane *even if the packets do not evoke a reply from hosts in the SDN*.

In contrast, previous SDN inference attacks could only determine whether *legitimate requests to servers in the SDN went through*

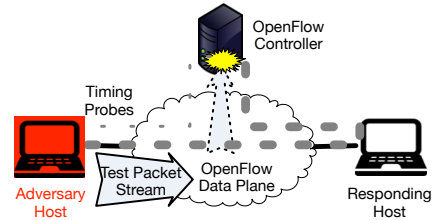


Figure 1: A summary of the control plane inference attack: an adversary sends probes that travel through the control plane while sending test packets into the network. If the test packets put load on the control plane, the probes will take longer to return.

the control plane. By overcoming this restriction, this new attack allows adversaries who control only a single host in an SDN to learn more about the rules in a network's forwarding table and the controller's logic. An adversary can use this knowledge to better plan subsequent stages of attacks. For example, an adversary can use our attack to learn short sequences of packets that cause the controller to install forwarding rules, then later send many such packet sequences to overload switch forwarding tables and degrade the entire network's performance.

In the remainder of this paper, we explain the inference attack in more detail and demonstrate its feasibility on a testbed with real OpenFlow hardware.

2. INFERENCE ATTACK OVERVIEW

Our threat model is based on an adversary that has root access to a single host in a network and wishes to learn as much information as possible about how the network operates without needing to compromise any other devices on the network, including the switches and controller. This model reflects two scenarios: first, an adversary performing a multi-staged attack that has just gained access to one host in a network through malware or social engineering and now wishes to plan subsequent stages; second, a user of a shared network or data center who wants to attack other users.

Figure 1 summarizes the attack. The adversary simultaneously sends a stream of *timing probes* and a stream of *testing packets* into the target network. The timing probes are pings to another server in the network that are specially crafted to travel through the control plane, and their round trip time (RTT) depends on the control plane's load. The test packets are spoofed packets that all have the same value for one or more packet header fields. If sending a stream of test packets causes the probe RTTs to increase, the adversary can infer that the control plane is processing the testing packets. By sending different test packet streams, the adversary can

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SDN-NFVSec'16, March 11 2016, New Orleans, LA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-4078-6/16/03...\$15.00

DOI: <http://dx.doi.org/10.1145/2876019.2876030>

Test Packet Streams:

$test_packet_stream ::= \langle template, size, transmit_rate \rangle$

Stream Templates:

$template ::= \langle header_field = field_value, \dots \rangle$

Header Fields:

$header_field ::= mac_source \mid mac_dest \mid ip_source \mid ip_dest \mid \dots$

Header Values:

$field_value ::= C$ (i.e. each packet in the stream will have the same constant value C for this field)
 $\mid *$ (i.e. each packet in the stream has a random value for this header)
 $\mid \vec{C}$ (i.e. the header field value for the i th packet in the stream will be $\vec{C}[i]$)

Figure 2: Syntax for test packet streams.

learn about the network. For example, if probe RTT is low when the adversary sends a stream of test packets with fixed source and destination MAC addresses, but high while the adversary sends a stream with the same destination MAC address but random MAC source addresses, the adversary can infer that the control plane processes packets with unknown source MAC addresses.

Timing Probes Timing probes allow an adversary to estimate how long the control plane takes to respond to a request and, in turn, how much load it is under. There are several ways to implement timing probes. One approach, which we test in Section 3, is to use spoofed ARP requests. In an OpenFlow network, MAC learning is often implemented in the control plane: an OpenFlow switch sends the first packet from each new MAC address to the controller, which figures out what rule to install on the switch to forward future packets to that address. If the adversary sends an ARP request to another host on the network using a spoofed source MAC address, the request and/or reply will be routed through the control plane, and the RTTs of ARP request/reply pairs can be used to time the control plane.

Alternately, if the OpenFlow controller is reachable from the data plane (i.e. there is no isolated control network or VLAN), the adversary can simply send an OpenFlow Echo message request to the controller. The controller will reply with an Echo response containing the original Echo message. The RTTs of echo request/response pairs can be used to time the control plane.

Test Packet Streams While measuring the control plane’s response time with timing probes, the adversary injects streams of test packets into the network. If the packets in a test stream end up being processed by the control plane, the RTT of the adversary’s timing probes will increase. By setting one or more packet header fields to the same value for all packets in a stream, the adversary can learn which classes of traffic get processed by the control plane.

For example, if an adversary wanted to learn whether there was a destination based forwarding rule installed in the data plane for a host with a MAC address of $0x0A$, they could send a test packet stream with packets that all had a MAC destination address of $0x0A$ and random MAC source addresses. If the RTT of the timing probes does not increase, the controller must not be processing the test stream packets, implying that a destination based forwarding rule is installed. On the other hand, if the timing probe RTT increases, the controller must be processing the test stream packets, implying that a destination based forwarding rule is not installed.

We specify test packet streams using the syntax in Figure 2. A test packet stream has: a *template* that specifies the header values of each packet in the stream; a *size* that specifies how many packets

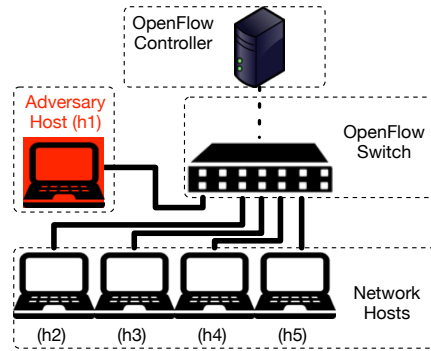


Figure 3: A diagram of our testbed network.

are in the stream; and a *transmit rate* that specifies how quickly the stream should be sent into the network in packets per second.

3. EVALUATION

In this section, we evaluate the inference attack on a physical OpenFlow testbed. We focus on the following questions:

- Can an adversary determine if test packets reach the controller?
- Can an adversary determine if the controller is installing forwarding rules in response to test packets?
- What higher level properties can an adversary learn about a network using this inference attack?

The Testbed Figure 3 illustrates our testbed network. It contains: a hardware OpenFlow switch, a Pica8 3290 with an Broadcom Firebolt-3 forwarding engine that processes packets in hardware according to OpenFlow rules, a 825 Mhz PowerPC CPU, and 512MB of memory, and runs Debian 7; a control server, a quad-core Intel i7 machine with 4GB of RAM, running Ubuntu 14.04 server LTS and the Ryu OpenFlow controller [1]. We connected 5 hosts to the switch, which we refer to as h_1 through h_5 . The hosts had MAC addresses of $00:00:00:00:00:01$ through $00:00:00:00:00:05$, IP addresses of $1.1.1.1$ through $1.1.1.5$, and were connected to the physical ports 1 through 5 on the switch, respectively. Each host was a dual-core Intel Core-2-Duo machines with 2GB RAM. All network connections (i.e. switch to controller and switch to host) were via gigabit ethernet.

The Adversary The adversary controlled host h_1 , and could send arbitrary raw packets into the network. To time the control plane, the adversary sends ARP requests to host h_5 at a rate of 4 per second, using the technique described in Section 2.

Controller Logic The OpenFlow controller runs a simple MAC learning application. When the network starts up, the controller installs a low priority rule into the switch that sends each packet up to the controller as an OpenFlow *packet_in* message that includes the ID of the port where the packet entered the switch. The control application keeps a map from MAC addresses to ports, which it fills using the source MAC address and input ports of the *packet_in* messages from the switch. When the controller receives a *packet_in*, it also check the MAC table for the destination MAC of the packet. If the table contains the address, the controller installs a rule onto the switch that forwards all packets with that MAC address out of the associated port. Otherwise, the controller instructs the switch to flood the packet.

Switch Logic To bootstrap the switch, we preinstalled the forwarding rules depicted in Table 1. This models a scenario where

Source MAC	Destination MAC	Action	Priority
*	00:00:....:02	Output on port 2	High
00:00:....:03	00:00:....:04	Output on port 4	High
*	ff:ff:....:ff	FLOOD	Medium
*	*	Send to controller	Low

Table 1: The testbed switch’s initial forwarding table. All packet header fields not shown are set to wildcards.

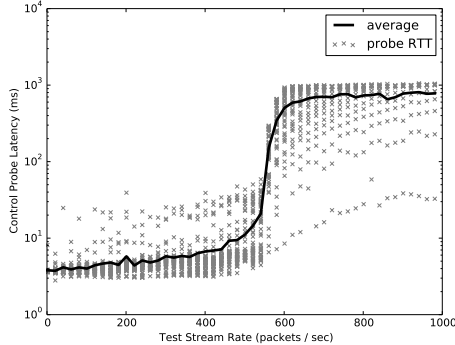


Figure 4: Timing probe RTT as test stream packet rate varies, for test streams that are processed by the control plane.

some forwarding rules are installed (either by the controller or by the network operator) *before* the adversary gains access to the host.

3.1 Are packets reaching the controller?

To determine the effect of packets reaching the control plane on probe RTTs, we measured probe RTTs while sending a test packet stream into the network from h_1 that matched only the default send to controller rule on the switch. We used the packet stream template: `<mac_source=00:00:....:0A, mac_dest=00:00:....:0E>`. Figure 4 shows probe RTT as the rate of the test stream varied in separate trials. There was a significant nonlinear relationship between probe RTT and test stream rate, with a jump from $<10\text{ms}$ to $>500\text{ms}$ when the stream rate went above 550 packets per second. Figure 5 shows probability density estimates for probe RTTs for different test stream packet rates.

There was a statistically significant difference between the means and variances of all three distributions, according to p-value tests. In our testbed, an adversary would be able to determine if the packets in a test stream were reaching the control plane by comparing the distribution of probe RTTs before sending the test stream with the distribution of probe RTT’s while sending the test stream.

Learning about Forwarding Tables An adversary could leverage this to learn about the rules installed in the switches’ forwarding tables. For example, in our testbed, the adversary can learn that there are forwarding rules installed to direct traffic from h_1 to h_2 but not h_3 or h_4 by measuring probe RTTs while sending no test packets, and then, in separate trials, measuring probe RTTs while sending test packet streams with templates:

```
<mac_source=00:....:01, mac_dest=00:....:02>;
<mac_source=00:....:01, mac_dest=00:....:03>;
<mac_source=00:....:01, mac_dest=00:....:04>;
```

Figure 6 shows the distributions of the RTTs for each test packet stream. The test packet stream to h_2 did not cause a statistically significant shift to the RTT distribution, indicating that the switch must have handled the packets without forwarding them to the controller. However, the test packet streams to h_3 and h_4 caused a statistically significant shift to the distribution, indicating that the packets did not match a rule and ended up in the control plane.

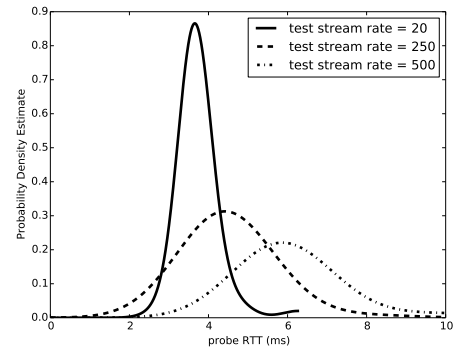


Figure 5: Probe RTT distribution estimates with test streams of different rates that get processed by the control plane.

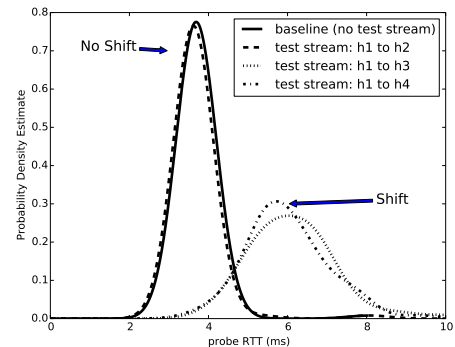


Figure 6: Probe RTT distribution shifts when the adversary sends test streams with no matching forwarding rules.

Once an adversary knows that a forwarding rule exists in the network, they can figure out which fields of the rule are wildcarded by sending test streams that each randomize one field of the base test stream that matches the rule. If the rule contains a wildcard for a field, the packet stream with that field randomized will not increase the timing probe RTT statistics. If the rule does *not* contain a wildcard for the field, the timing probe RTT statistics will increase.

Table 2 shows probe RTT statistics while randomizing different fields of test stream templates that match forwarding rules on the switch in our testbed. The probe RTT and variance remains low when the `mac_source` field of the first test flow is randomized, which indicates that the rule forwarding traffic to `00:....:02` must have its `mac_source` field wildcarded. Randomizing any other field in either stream results in a higher probe RTT and variance, which indicates that the forwarding rules must require exact matches in those fields.

Example Use: Attack Planning If a network’s forwarding rules match on both the source and destination, an adversary can build a communication graph for the hosts in the network by checking for rules that forward packets between each pair of hosts. Hosts that have many edges in the communication graph (i.e. communicate with many other hosts) may be critical to the network and an ideal target for DoS attacks, while hosts with few edges may be less frequently used and an ideal target for the adversary to infect while minimizing disruption to the network.

3.2 Is the controller installing forwarding rules?

To determine the effect of the controller installing a forwarding rule on the switch, we measured probe RTTs while sending a sequence of two test packet streams that cause the MAC learner

base template: $\langle \text{mac_source}=00:\dots:01, \text{mac_dest}=00:\dots:02 \rangle$		
field randomized	probe RTT average	probe RTT variance
mac_source	3.64	0.46
mac_dest	12.11	11.57
base template: $\langle \text{mac_source}=00:\dots:03, \text{mac_dest}=00:\dots:04 \rangle$		
mac_source	11.37	9.31
mac_dest	10.62	9.11

Table 2: Probe RTT statistics while sending test streams with different packet header fields randomized.

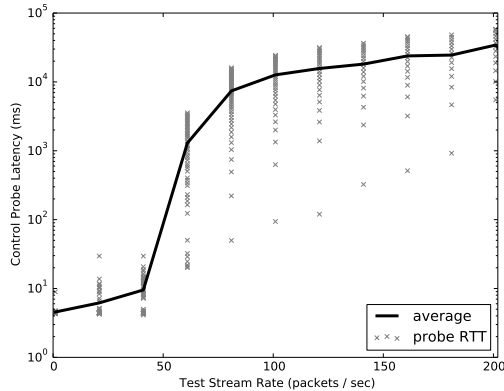


Figure 7: Probe RTT as test stream packet rate varies, for a test stream where each packet invokes a rule installation.

on the controller to install one rule for each packet in the second test stream. More specifically, we generated two vectors of MAC addresses: \vec{S} and \vec{D} , each of which had 1000 unique random MAC addresses. We then sent a stream with the template $\langle \text{mac_source}=\vec{S}, \text{mac_dest}=\vec{D} \rangle$. The switch forwarded each packet of the stream to the controller, which added the `mac_source` to its local table that maps mac addresses to physical ports and then instructed the switch to flood the packet. Next, we send a stream with the reverse template: $\langle \text{mac_source}=\vec{D}, \text{mac_dest}=\vec{S} \rangle$. The switch also forwarded each of these packets to the controller. However, since the controller knew the location of each destination mac address (due to the first stream), it installed a forwarding rule onto the switch for future packets with that destination mac.

Figure 7 shows the RTT of the timing probes during the second test stream, as test stream rate varies. Each packet in this stream caused the controller to install a forwarding rule. These test streams shifted probe RTT statistics much more than test streams that caused the controller to process packets without installing rules (*i.e.* those depicted in Figure 4). Even a slow test stream that caused 60 flow rules per second to get installed increased the probe RTT by approximately 100ms, for example.

An adversary on our testbed would thus be able to not only determine if test packets are being processed by the control plane, *but also whether or not the control plane was installing rules in response to those packets.*

Learning about controller logic An adversary could leverage this to learn about the controller’s logic. For example, in our testbed, an adversary can learn which sequence of packets causes the controller to install a rule that forwards traffic from a `MAC src` to `dest` by performing trials that test different sequences of packet streams. Table 3 shows two such trials. In each trial, the adversary sends two low rate test streams that may or may not trigger rule installations, and then one high rate test stream to determine if rules were installed. In the first trial, the latency for the third stage is statistically higher, indicating that the first two stages could

Stream Template	Rate	RTT average	RTT variance
Trial 1:			
$\langle \text{mac_source}=\vec{S}, \text{mac_dest}=\vec{D} \rangle$	50	4.09	1.57
$\langle \text{mac_source}=\vec{S}, \text{mac_dest}=\vec{D} \rangle$	50	4.22	2.05
$\langle \text{mac_source}=\vec{S}, \text{mac_dest}=\vec{D} \rangle$	500	12.72	11.59
Trial 2:			
$\langle \text{mac_source}=\vec{D}, \text{mac_dest}=\vec{S} \rangle$	50	4.11	1.71
$\langle \text{mac_source}=\vec{S}, \text{mac_dest}=\vec{D} \rangle$	50	812.93	519.88
$\langle \text{mac_source}=\vec{S}, \text{mac_dest}=\vec{D} \rangle$	500	4.32	1.58

Table 3: Probe RTT statistics during two trials to determine the sequence of packets that causes a rule to be installed. High RTT followed by low RTT indicates that rules are installed.

not have caused rules to be installed. In the second trial, however, the latency during the third stage is low, despite the high rate of the stage. This indicates to an adversary that the first two stages must have triggered rule installations that match packets in the third stage’s stream; the high latency in the second stage confirms that the controller is actively installing rules during that stage. Thus, the adversary can infer that when the controller observes a packet from $x \rightarrow y$ followed by a packet from $y \rightarrow x$, it will install a rule forwarding $y \rightarrow x$.

Example Use: DoS Attacks OpenFlow switches store forwarding rules in TCAM memory that allows them to match packets quickly. TCAM memory is expensive and power hungry, so modern OpenFlow switches can only store up to approximately 10,000 forwarding rules in TCAM. Any additional rules are put into tables in standard memory, and matching packets against these rules is much slower. If an adversary learns the sequence of packets that causes the controller to install a rule, they can saturate TCAM flow tables with a small number of packets and greatly degrade the network performance of all hosts.

4. CONCLUSION

Control plane inference attacks allow adversaries to learn about a SDN network without compromising its infrastructure. We developed a more advanced inference attack based on measuring the control plane’s load, demonstrated that it is effective on real OpenFlow hardware, and provided examples of how it can help adversaries stage more advanced attacks. As SDN adoption grows, we believe that it is important to evaluate the potential of inference attacks in real networks and work to develop practical defenses.

Acknowledgements We wish to thank the anonymous reviewers for their input on this paper. This research was partially supported by NSF SaTC grant numbers 1406192, 1406225, and 1406177.

5. REFERENCES

- [1] Ryu. <http://osrg.github.io/ryu/>.
- [2] R. Kloti, V. Kotronis, and P. Smith. Openflow: A security analysis. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–6. IEEE, 2013.
- [3] 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.
- [4] S. Shin and G. Gu. Attacking software-defined networks: A first feasibility study. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 165–166. ACM, 2013.