Timing-based Reconnaissance and Defense in Software-defined Networks

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

Anurag Dubey
University of Colorado Boulder
anurag.dubey@colorado.edu

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

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

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

ABSTRACT

Software-defined Networking (SDN) enables advanced network applications by separating a network into a data plane that forwards packets and a control plane that computes and installs forwarding rules into the data plane. Many SDN applications rely on *dynamic* rule installation, where the control plane processes the first few packets of each traffic flow and then installs a dynamically computed rule into the data plane to forward the remaining packets. Control plane processing adds delay, as the switch must forward each packet and meta-information to a (often centralized) control server and wait for a response specifying how to handle the packet. The amount of delay the control plane imposes depends on its load, and the applications and protocols it runs. In this work, we develop a non- intrusive *timing attack* that exploits this property to learn about a SDN network's configuration. The attack analyzes the amount of delay added to timing pings that are specially crafted to invoke the control plane, while transmitting other packets that *may* invoke the control plane, depending on the network's configuration. We show, in a testbed with physical OpenFlow switches and controllers, that an attacker can probe the network at a low rate for short periods of time to learn a bevy of sensitive information about networks with $> 99\%$ accuracy, including host communication patterns, ACL entries, and network monitoring settings. We also implement and test a practical defense: a *timeout proxy*, which normalizes control plane delay by providing configurable default responses to control plane requests that take too long. The proxy can be deployed on unmodified OpenFlow switches. It reduced the attack accuracy to below 50% in experiments, and can be configured to have minimal impact on non-attack traffic.

1. INTRODUCTION

Motivation. Software-defined Networking (SDN), a paradigm for programmable networks, has become increasingly popular. Organizations including Facebook [7], Google [20], and the NSA [2] have deployed large scale SDN networks with many other organizations planning future deployments. SDN's popularity stems from its ability to serve as a platform for innovative network applications that provide many benefits such as cost reduction [20], energy savings [18], and improved security [12].

SDN radically changes network design, replacing fixed functionality elements (*e.g.* switches, routers, firewalls, address translators) with generic network elements (*i.e.* the data plane) that forward packets at high speeds by matching them against simple forwarding rules. A centralized control server (the control plane) manages and updates the rules on switches based on the functionality required by the network.

Controllers, in particular, may perform more advanced packet processing as needed, especially when packets arrive at a switch that does not have a matching rule installed. In those situations, the packet information is forwarded from the data plane switch to the control plane controller for further processing. The controller makes a forwarding decision for the packet and optionally assigns new rules to the switch to handle further packets of the same pedigree based on specific source-destination-port information or on pattern matching, such as on sub-net information.

While this design clearly offers tremendous benefits and flexibility to network operators, it also comes with new security challenges. *One important challenge, the focus of this paper, is ensuring that the elements of an SDN do not leak sensitive network configuration or usage information.*

Previous work has demonstrated that SDN networks have timing side channels based on the difference in the amount of time it takes two hosts to establish a connection, that is, if the connection time for a flow is high, the first packets of that flow likely invoked the control plane. These side channels may reveal if a network runs OpenFlow [33], the size of switches' forwarding tables [29], as well as whether links contain aggregate flows [23].

Here, we extend this type of attack so that it both reveals more sensitive network information and is stealthier. Further, we present a deployable and effective defense to this style of timing attack.

Our work. In this paper, we develop a more sophisticated timing-based side channel attack that can be launched by an adversary with access to only a single machine on the target network. Using the attack, the adversary can learn many more details about a network configuration than reported in prior work, without initiating connections to other hosts in a network. Much of the revealed information would be considered highly sensitive, including host communication records, network access control configurations, and

ACSAC '16, December 05 - 09, 2016, Los Angeles, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4771-6/16/12…$15.00
DOI: http://dx.doi.org/10.1145/2991079.2991081

network monitoring policies.

At a high level, the attack works by estimating a control plane's load by injecting *timing pings* into the controlled network. These pings are specially crafted such that the switch must invoke the control plane to forward them. At the same time, the attacker sends a second stream of *test packets* into the network. By comparing the turnaround time of the pings to previously collected baseline samples, the attacker can deduce whether the test packets were processed by the controller or simply forwarded by the data plane, and further, determine whether the controller installed new rules in response to the test packets.

With a few trials using different test streams, an attacker can learn which flow rules are installed in the data plane and which sequences of packets cause the control plane to install new rules. Depending on the SDN application, this attack may also reveal security sensitive details about the network, such as:

(i) the **host communication graph** of which devices in the network communicate with each other (if the network is running a common layer 2 MAC learning application);

(ii) the **access control lists** of the switch specifying which traffic flows the switch is configured to drop (if the network is running an access control application);

(iii) and the usage of **monitoring and packet counting rules** to collect flow records on the switch (if the network is running a monitoring application).

Evaluation. To concretely evaluate the feasibility of these attacks, we experimented using a physical OpenFlow testbed with real SDN hardware and background traffic (prior work [33, 29, 23] used virtual network environments and virtual machines).

We find that the attacker can determine which role the controller plays in processing test packets with extremely high accuracy ($>$ 99%), even with very low rates (10 timing pings and 500 test packets per second) and short trials (< 5 seconds). We determine that bottlenecks in the control plane are the root cause of the attack's effectiveness, as they add statistically significant delays to the timing pings ($p < 0.005$ using a t-test) that vary depending on how the control plane processes test packets. We show the attack can reveal sensitive information about more complex control applications with nearly perfect fidelity (such as ACL or flow counting).

Defense. To defend against this new attack, we propose the use of a *timeout proxy* that sends a default forwarding instruction to the data plane if the control plane fails to respond within a fixed time period. We implemented this defense as a software application that can run on the switch itself (*i.e.* below any control plane bottlenecks), as part of an existing OpenFlow environment. Experimenting with the proxy in our testbed, the differences in control plane load measurements became unobservable to the timing attack, and reduced its accuracy to $< 50\%$. The proposed proxy is extremely robust and can handle up to approximately $10,000$ requests per second, and if deployed on a machine dedicated to defense (not a switch), it could easily handle considerably higher rates of attack.

Contributions. To summarize, this paper's contributions are:

(i) A novel SDN timing attack that can induce a side channel for learning about OpenFlow networks and reveals more information than previous methods.

(ii) A detailed experimental analysis of this timing attack on a testbed with physical OpenFlow hardware and real background traffic, the first evaluation of SDN timing attacks on physical hardware.

(iii) An effective software based defense against the timing side channel that can be deployed either directly to physical OpenFlow switches, or to dedicated middlebox servers, the first implemented and tested defense against timing side channels on OpenFlow networks.

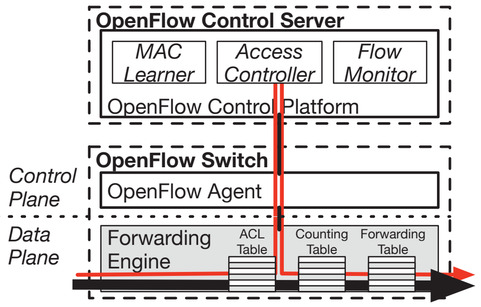

Figure 1: OpenFlow divides a network into a *data plane*, that forwards packets quickly using simple tables, and a *control plane*, that manages the data plane using more complex actions.

2. RELATED WORK

Side channel attacks. There is a large body of work on using side-channels to leak secret information from computer systems, most prominently for recovering secret cryptographic keys [24, 22, 37]. Many side-channel attacks are based on *timing*; an adversary analyzes the timing of execution [25] or caches [37] to learn details about the code that is executing and what data it is operating on. Our timing-based side channel attack is similar to previous work that times *execution*, although execution timing has not previously been applied to the SDN domain.

Previous work has demonstrated that timing-based side channel attacks can be used remotely [11], to recover details about a host's operating system [26], expose private web pages [9], and to recover cryptographic keys [10] across local and wide area networks. Our attack is designed to work across a local network, though based on these previous studies and our tests, it seems likely that with more advanced techniques, future SDN timing attacks could also be effective against wide area SDN deployments, as they become more widespread. [20, 17].

OpenFlow side channel attacks. A SDN network is topologically similar to a traditional computer network, but behaviorally much more complex, with many different components that could be targeted with a side channel attack. OpenFlow [30] is the de-facto standard for SDN: it defines a packet processing model, API, and remote management protocol for switches. Figure 1 summarizes the architecture of an OpenFlow Network, which can be divided into a *data plane* and a *control plane*. The data plane processes most of the packets that a switch forwards by matching packet headers against rules in flow tables that define simple actions (the green line in Figure 1). The control plane processes packets that the switch does not know how to handle, and can install new flow rules into the data plane for future packets (the red line in Figure 1). The data plane is implemented as a highly optimized *forwarding engine* in the switch. For physical switches, this is implemented with specialized hardware. For virtual switches [32], the forwarding engine is often a kernel module. Forwarding engines do not directly im-

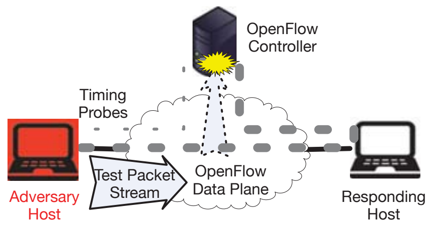

Figure 2: Summary of our control plane timing attack: an adversary times the control plane's execution while sending test packets into the network. If the test packets put load on the control plane, the attacker will observe longer response times with the timing probes.

plement the OpenFlow API. Instead, an OpenFlow Agent, which runs as software on the switch (even physical switches), translates OpenFlow messages from the control server into a lower level forwarding engine API. The control server implements more complex logic, such as MAC learning when a new device is connected to the network, managing the access control policies of all switches in the network, or configuring flow monitoring rules on switches.

There has been little research into applying timing-based side channel attacks to Software-defined networking. To our knowledge, there have only been three previous works in this area. Kloti et al. [23] developed a simple attack which measures the set up time of sequential TCP connections to determine if an SDN is using forwarding rules that aggregate TCP connections. Shin and Gu [33] proposed a SDN scanner that measures the response time of pings to determine whether or not a network is using SDN. More recently, Leng et al. [29] designed an attack that measures the response time of requests to determine the approximate capacity of an OpenFlow switch's forwarding table (*i.e.* how many flow rules it can store). All of these attacks can be viewed as *cache-timing* attacks, where the switch's flow table is the cache and the goal is to determine if a legitimate request is processed by the cache or not. In contrast, our attack times *control plane execution*, where the goal is to determine if *arbitrary* packets crafted by the attacker get processed by the control plane. Our approach reveals additional, more security sensitive information about an OpenFlow network.

Additionally, the existing literature on SDN timing attacks uses software based networks, in the Mininet environment [28] with virtual switches [32]. In practice, hardware based OpenFlow switches manufactured by vendors such as Broadcom, Cisco, Juniper, Brocade, and others are widely deployed. These physical devices have vastly different architectures and performance characteristics compared to virtual OpenFlow switches: latencies and packet forwarding rates are orders of magnitude better [6]; but flow installation rate and flow table capacity are orders of magnitude worse [8]. In this work, we perform extensive testing with physical OpenFlow switches, and provide a first look at the effectiveness of timing attacks with a new degree of experimental realism.

attacks. Kloti et al. [23] speculate that there are likely to be detection or randomization based defenses against SDN timing attacks and Leng. et al. [29] briefly suggests further research into attack detection and automated flow table maintenance. We have taken the next step by designing a defense against the timing attack, implementing it as software that runs on a physical OpenFlow switch, and experimentally demonstrating its effectiveness.

3. CONTROL PLANE TIMING ATTACK

Figure 2 summarizes our control plane timing attack. The attacker learns about the network by performing trials in which they send a stream of *timing probes* and a stream of *testing packets* into the target network, and then comparing the timing probe RTTs to a previously collected baseline sample to learn some details about the network's configuration. We assume that the attacker has control of a single host on the network and can inject packets into the network at will, but does not want to disrupt network performance in an end-user detectable way.

An attacker performs a series of trials. In a trial, an attacker: (1) collects a baseline round-trip- time (RTT) measurement while transmitting a stream that has a known impact on the control plane; (2) transmits a test stream to get a sample RTT measurement; and then, (3) compares the two RTT measurements. Based on the comparison (using a simple t-test) of the two distributions of RTT measurements, the attacker then attempts to deduce whether the control plane installs new flow rules into the data plane in response to the test packets or if other control level actions were taken.

The timing probes are request packets to hosts or devices on the network that cannot be forwarded without invoking the control plane. Timing probes essentially act as *control plane pings*, and their RTT value informs the attacker of how long the controller takes to process packets at particular points in time.

The test packets are spoofed packets that all have the same value for one or more packet header fields to control for the kind of inference the attacker wishes to perform. Depending on how the timing probe RTT changes when transmitting a testing stream, the attacker can infer the role that the control plane plays in handling packets with those header values. By performing repeated trials with different test packet streams, the adversary can discover more and more information about the network configuration.

3.1 Threat Model

The threat model assumes the adversary has control of a host in the network, and seeks to learn about the network without needing to compromise any of the network infrastructure or other hosts. This threat model matches two real world scenarios where an attacker may wish to learn information about the network configuration.

First, an intelligent adversary performing a complex, multi-staged attack who has just gained access to one host in the target network through malware or social engineering and now wishes to plan subsequent stages. For example, consider the recent Target data breach [21], where attackers installed malware on point of sale terminals to collect credit card numbers; or the more recent breach at Juniper [19], where adversaries installed back doors into source code stored deep within Juniper's network. In scenarios like these, the ability to learn sensitive information about the network in a reconnaissance could greatly benefit attackers.

A second scenario that fits this model is a malicious user of a shared network (such as at a data center or cloud provider) that may wish to learn about other users of the network via the network configuration settings of the switch. As we will show, the timing attack is able to deduce host communication pairs as well as collect information about SDN monitoring applications.

3.2 Timing Probes

Timing probes are specially crafted request packets that the adversary sends into the network to learn how long the control plane takes to process packets at specific points in time. Timing probes must have three properties: first, they must evoke a response from a device in the network, so that the adversary can compute a RTT for each timing probe; second, they network must not be able to

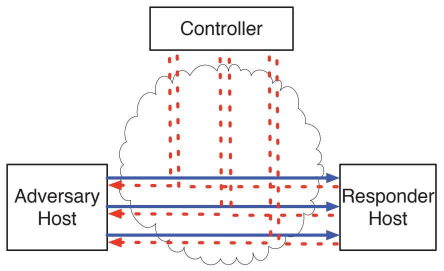

Figure 3: Timing the control plane with spoofed ARP requests to a one host.

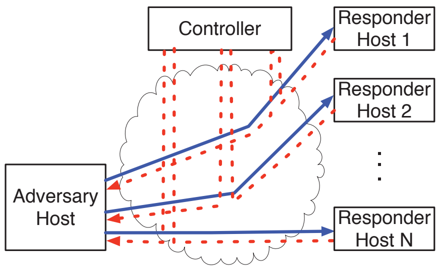

Figure 4: Timing the control plane with legitimate ARP requests to multiple hosts.

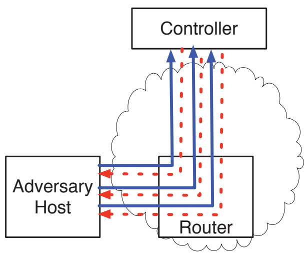

Figure 5: Timing the control plane with low TTL IP packets.

forward the packet without invoking the control plane, so that controller processing time is a factor in the RTTs of the timing probes.

Many types of requests can potentially act as timing probes, depending on the configuration of the targeted network. For example, in a naive network that sends all packets to the control plane, a TCP syn packet to any legitimate host in the network would be a timing probe.

Below, we describe three types of timing probes: two timing probes that can be used on a Layer 2 network (*i.e.*, where switches forward based on MAC address), and one that can be used on a Layer 3 network (*i.e.*, where switches forward based on IP address).

Layer 2 Timing Probe: Spoofed ARP Requests Layer 2 networks, both traditional and SDN, typically forward packets using *MAC learning*, a simple algorithm to associate MAC addresses with physical ports on a switch. Whenever a switch receives a packet from a device with a MAC address that is not present in its forwarding table, the switch sends the packet to the controller, which saves the (input port, MAC address) pair. When the controller receives a subsequent packet destined for that device, it instructs the switch to forward the packet out of the port where the device was observed, and installs a corresponding rule into the switch's forwarding engine. Traditional networking equipment used flow rules that mapped a destination address to an output port on the switch. Most OpenFlow implementations use finer grained flow rules that specify both the source and destination address because it provides the controller with greater visibility and allows additional path optimization techniques [1].

In networks that use MAC learning, an adversary can use ARP requests as timing probes. ARP is widely used protocol for mapping IP addresses to MAC addresses. The adversary selects a host on the network that uses ARP, then sends it ARP requests with spoofed source MAC addresses. Figure 3 depicts what will happen when the adversary sends the ARP request packets into the network. The ARP requests themselves will not invoke the control, since ARP requests are broadcast, which a forwarding engine can do without invoking the control plane. However, the ARP *reply*, which will have the randomly generated address as its destination MAC, *will* need to invoke the control plane because the switch's forwarding engine will not have a rule mapping the randomly generated address to a port.

It is important to not overload the network with spoofed MAC addresses as this might be noticeable (and detectable) to a network monitor. It may also degrade the performance of the network, something the attacker wishes to avoid as it may affect the ability to infer information from the network. Fortunately, timing probes are useful even at *extremely* low rates, because each probe provides a useful measurement and just a few is sufficient to notice variations in RTT rates when a testing stream is introduced. For example, in all of our experiments in Section 4 a timing probe at a rate of 10 per second was ample for inferring sensitive information. On a large network with many Layer 2 connected devices (as in a data center), there should be a reasonable amount of ARP traffic such that timing probes will cause minimal interference with normal network activity.

[1]In a traditional, pre SDN network, the switch's CPU runs the controller level MAC learning logic.

Layer 2 Timing Probe: Legitimate ARP Requests In some layer 2 networks, it is not possible to send packets into the network with random MAC source addresses due to network access control systems that either drop packets from MAC addresses that do not belong to pre-authorized devices or limit the number of devices that can be connected to each physical port on a switch or router [13].

Figure 4 illustrates a technique to generate ARP based timing probes that works around this defense by taking advantage of *forwarding rule timeouts*: switch forwarding tables have limited memory and, as a result, are usually programmed to delete forwarding rules if they have not been used for more than a threshold period of time. To use this approach, the attacker selects a set of N hosts to act as ARP responders and sends them each one legitimate ARP request, in sequence. Assuming all the rules that forward packets from the ARP responders to the attacker have timed out, each of the ARP replies will need to be processed by the control plane so it can re-install the forwarding rules.

An adversary would not need a large number of responder hosts to use this approach because, as our evaluation demonstrates in Section 4, timing probes only need to be sent at a very low rate and for only a short amount of time (*i.e.*, 10 per second for approximately 5 seconds).

Layer 3 Timing Probe: Low TTL Packets SDNs can also be configured to forward at the IP level (*e.g.*, programming OpenFlow switches to act as routers [5]). This allows an adversary to use IP packets with low TTL values as timing probes. OpenFlow switch cannot generate ICMP packets in their forwarding engines and must send packets with expiring TTL values to the controller so that it can generate the correct response. As Figure 5 depicts, when an adversary sends IP packets with TTL=0 into the network, the first network element configured to act as a router will send the packet to the controller, which will generate the appropriate ICMP response and send it back to the adversary's host.

3.3 Test Packet Streams

A test packet stream is a sequence of packets with a small invariant that the attacker sends into the network at a constant rate to determine if the data plane needs to invoke the control plane for the given invariant. If the control plane is invoked that provides information for some configuration of the network. We define a test packet stream with a template for an invariant selection that specifies either a constant or randomly selected value for the Ethernet, IP, and TCP/UDP header fields of each packet.

The attacker can determine the role that the controller plays in forwarding the test packets by analyzing the RTTs of timing probes that are ongoing during the testing stream. Based on the measurements from the RTT the following deductions could be made:

(i) If the data plane *contains a rule* that matches the packets in the test stream, it will place no additional load on the controller, and the timing probe RTTs will be low during the test stream.

(ii) If the data plane *does not contain a rule* that matches packets in the test stream, the packet will be forwarded to the controller placing a moderate load on the control plane, and introducing some delay to the timing probes.

(iii) If the data plane *does not contain a rule* that matches the forwarding packets in the test stream, and the control plane *installs new forwarding rules* into the data plane in response *to each packet* in a test stream, there will be a heavy load placed on the control plane as flow rule installation is an expensive operation[2], adding much more latency to the timing probes than the previous scenario.

3.4 Detecting RTT changes

An attacker could learn what effect a test stream has on the network by comparing the distribution of probe RTTs observed while transmitting the stream with a sample of *baseline* RTTs representing a known effect. A simple method for this that we found effective is a straightforward application of the Student's t-test [36], a statistical test that compares if two samples are drawn from the same distribution. A t-test produces a t-statistic that measures the size of the difference between the samples relative to variation in the sampled data. A t-statistic can be converted into a p-value, which speaks to the likelihood that the two samples share the same distribution. Typically, a p value less than 0.05 is considered an indicator of statistically significant difference between the samples in that they are truly measuring different distributions.

One baseline stream is simply a list of Ethernet packets with source and destination addresses that are chosen at random, without repetition. The controller will process each packet in this stream, to perform MAC learning, but will not install a rule since it will never receive a packet destined for that address. By using a t-test to compare the RTTs observed while transmitting a test stream to the RTTs observed while transmitting this baseline stream, an attacker can conclude:

(i) If the p-value is high, the timing probe RTTs likely share the same distribution as the baseline sample, so like the baseline stream, the test stream is likely processed by the control plane, but does not cause additional flow rule installations.

(ii) If the p-value is low and the t-statistic is negative, then the timing probe RTTs likely have a different distribution than the baseline samples and a much *lower* mean, so the test stream packets are likely not processed by the control plane.

(iii) If the p-value is low and the t-statistic is positive, then the timing probe RTTs likely have a different distribution than the baseline samples and a much *higher* mean, so the test stream packets likely caused additional flow rule installations.

[2]Many OpenFlow devices only support a flow installation rate of <100 per second [8].

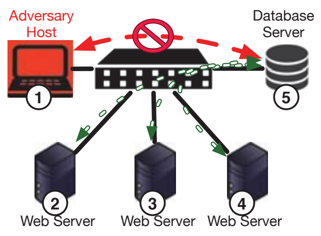

Figure 6: An example application of the timing attack: an adversary can learn which devices communicate with a back-end database server that the adversary cannot directly connect to, due to firewalling.

Priority	Source IP	Destination IP	Policy
10	1.1.1.2	1.1.1.5	**ALLOW**
10	1.1.1.3	1.1.1.5	**ALLOW**
10	1.1.1.4	1.1.1.5	**ALLOW**
1	*	1.1.1.5	**DROP**
DEFAULT	*	*	**ALLOW**

Table 1: An example ACL policy installed into the switch in Figure 6, which blocks connections to the database from all hosts besides the web servers.

3.5 Example Attack Scenarios

Learning host communication patterns. An attacker can learn if two devices (A and B) have recently communicated with each other by performing a trial that tests the presence of a forwarding rule that handles traffic between A and B. To do so is a straightforward application of the procedures described previously: the attacker records a baseline RTT and then measures the RTT after injecting a test stream with spoofed host address for A with destination B. If there is not a significant change in timing information, then controller must not be involved in forwarding, and there is likely a rule installed.

Learning host communication pairs may reveal broader information about the network that is especially useful in a multi-staged attack. It can help the adversary understand the purpose of devices that cannot be directly probed (*e.g.* due to firewalling).

Consider the scenario depicted in Figure 6. The adversary's goal is to deduce which device is the database server for the networks web server. The adversary can collect the MAC addresses of all devices on the network by monitoring ARP requests, and open TCP connections with the web servers (#'s 2 through 4 in Figure 6) in the network. However, the adversary cannot open TCP connections with the database server (#5 in Figure 6) because the switch drops all TCP packets destined for the database unless they come from a web server. The adversary can determine that server 5 may be a database server by using our timing attack to learn that there are rules installed on the switch that forward traffic from each of the web servers to server 5.

Learning ACL entries. One common application of OpenFlow is to provide access control, similar to a traditional firewall. This is generally implemented by configuring a switch to sequentially match packets against two tables: first, an access control table that can drop packets by matching them against entries specifying Layer 2 through 4 headers; second, a forwarding table that uses a standard algorithm, such as MAC learning, to select which port to send packets out of.

For example, the ACL policy depicted in Table 1 could be used in the example scenario in Figure 6 to block connections to the database server unless they originate from one of the network's web servers. An adversary can learn that these entries exist in the ACL by performing trials of the attack with test packet flows that use spoofed addresses, as follows:

1. First, the adversary chooses a MAC and IP source and destination addresses that do not exist in the network and performs an attack trial using test packets with those addresses. The packets will not match any entries in the ACL or forwarding table, and the switch will send them to the control plane for a forwarding decision, causing a measurable **high RTT** skew.

2. Next, the adversary performs trials with the *same* MAC addresses but *different* IP address pairs. If the IP address pair is blocked by the ACL, the adversary will observe a measurably **low RTT** in that trial (compared to the previous) because the ACL table will drop the packets before they reach the MAC table.

   If the IP address pair is blocked by the ACL, the adversary will continue to observe the **high RTT** values from the first trial, as the packets will pass through the ACL to the MAC learning table that sends them to the controller.

Learning about monitoring controllers. OpenFlow networks are capable of running security monitoring applications alongside their forwarding logic, as proposed in [34], [35] and [16]. These applications are generally implemented with a *counting table* that simply counts the number of packets and bytes from each IP address or flow. When a packet matches a flow in the counting table, the flow's counter is incremented and the packet is sent to the next table in the forwarding engine's pipeline (*e.g.* the forwarding able). When a packet does not match an entry in the forwarding table, the data plane sends the packet up to the controller which installs the appropriate counting rule. Periodically, the controller polls the forwarding engines for statistics about all of the flows in the counting table and performs analysis on the aggregated data.

An adversary can learn at least two important details about an OpenFlow monitoring application using our timing attack: whether the network is monitoring at the host or flow level and how frequently the controller receives updates from the switch.

To determine if a network is monitoring at the host or flow level, an adversary can build an *all pairs test stream* by generating a list of N random IP addresses then adding one packet to the test stream for each pair of IP addresses (*i.e.* N^2 packets). The adversary runs an attack trial with this test stream. If the network is not monitoring, the timing probe RTT will not shift when the all pairs test stream is transmitted because the control plane will not install any new rules due in response to the test stream. If the network is monitoring at the *per host* level, the probe RTTs will shift proportionally to $\frac{N}{test\_stream\_rate}$ because the controller will install a counting rule for each new IP address. Finally, if the network is running a *per flow* monitoring application, the probe RTTs will shift proportionally to $\frac{N^2}{test\_stream\_rate}$, the number of unique *flows* the adversary sent into the network per second, because the controller will install a counting rule for each new IP flow.

To determine how frequently the controller polls the switch for updates, the adversary can simply measure the timing probe RTTs over a longer period of time after running the trial described above. Whenever the controller polls the switch for statistics about the counting flows, it will place load on both the switch and controller, temporarily increasing the probe RTT.

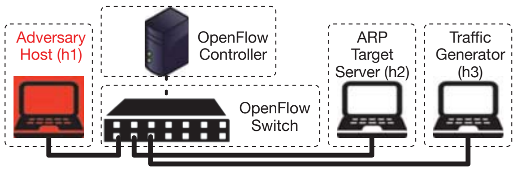

Figure 7: A diagram of our testbed network.

4. ATTACK EVALUATION

To evaluate the effectiveness of the timing attack, we performed experiments on a physical OpenFlow testbed with real OpenFlow equipment and background traffic. The evaluation is divided into three parts. First, we measured the accuracy of our attack at inferring changes in the network performance rates. Next, we performed benchmarks to understand the low level details of our attack: the effects of different attack rates, the impacts of background traffic, and determine bottlenecks in the control plane. Finally, we tested the techniques described in Section 3 as they are used to learn security sensitive details about a network.

4.1 Testbed and Background Traffic

The testbed. Figure 7 illustrates our testbed network. It contains: a hardware OpenFlow switch, a Pica8 3290 with a 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 [4]. We connected three hosts to the switch: h_1, the adversary controlled host; h_2, the host that the timing probe sends ARP requests to; and h_3, a host that replays background traffic into the testbed. Each host has 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.

Background traffic. To model real network conditions, we replayed a background trace from NCCDC 2015 [1], a three day cyber-defense competition in which 10 teams compete by defending networks composed of real devices against human adversaries. Each team's network contained 8 servers and 6 workstations running a mix of Windows, Linux, and BSD, 1 VoIP phone, 1 Juniper EX2200 switch, and 1 Juniper SRX210 gateway. The 10 teams were all connected to a core switch, also a Juniper EX2200. The Juniper switches are an equivalent class of hardware as our Pica 8 3290 switch, but do not use OpenFlow. The trace contains all the full, unanonymized packets that passed through the core switch. On average, the trace had a throughput of approximately 80Mbps and 10,000 packets per second. The traces are publicly available [3].

Attack parameters. Unless otherwise noted, we used the *spoofed ARP timing probes* described in Section 3.2 with a rate of 10 probes per second, a test stream rate of 500 packets per second, and a test stream duration of 5 seconds.

4.2 Attack Effectiveness

To measure the accuracy of the attack in trials we used the t-test technique described in Section 3 to determine whether a test stream was *processed by the dataplane*, *processed by the control plane*, or *caused new flow installations*. The experiment was executed over 300 trials in which each scenario was equally represented by test streams and compared against a baseline sample of RTTs collected

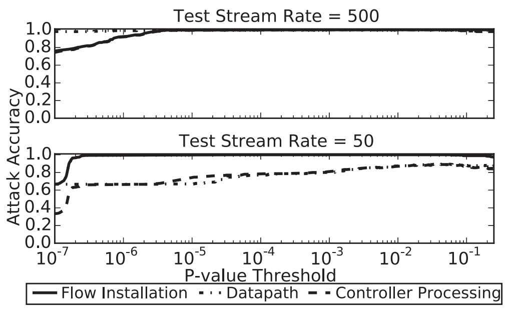

Figure 8: Attacker accuracy in predicting the control plane's role as p-value threshold varies.

while transmitting a test stream that caused control plane processing, collected prior to the trials.

Figure 8 shows the attack's accuracy in these trials, as we vary the p-value threshold used to distinguish between statistically significant RTT samples. At a test flow rate of 500 packets per second, the attack correctly classified *all* test streams with $> 99\%$ accuracy for a large range of thresholds ranging from .1 to .000001. At a much lower test flow rate of 50 packets per second, the attack had difficulty distinguishing between data plane processing and control plane processing.

However, due to the high load that flow installation places on the control plane, it was still possible to correctly identify whether or not a test stream caused new flow installations with a p-value threshold in the .1 to .000001 range. This suggests that if an attacker wishes to minimize their impact on the network, they can first send a test stream into the network at a low rate, to determine if it causes flow installations, and then when they are confident that it does not, send test streams at a higher rate to learn if it is processed by the control plane or passes through the data plane without controller interaction.

Given this wide range of thresholds, it would likely be straightforward for an adversary to calibrate the attack to a target network. A common p-value to indicate significance in many domains is .05, which was well within the range of thresholds that had very high accuracy in our trials. An adversary could also estimate a range of thresholds that yield high accuracy by measuring the difference, in terms of p-value, between RTT samples taken while transmitting a test stream known to be processed by the data plane vs. RTT samples taken while transmitting a test stream known to be processed by the control plane.

4.3 Attack Fundamentals

In this Section, we study at the attack at a lower level to better understand the factors that make it work.

Test stream rate. Figure 9 shows probe RTTs as the rate of the test stream varied in trials where the controller was configured to make a forwarding decision for each test stream packet. Figure 10 shows RTTs in trials where the controller also installed a rule in response to each packet in the test stream. Figure 11 shows an alternate view, empirical distributions of timing probe RTTs in different scenarios for test stream rates of 50 (left) and 500 (right) packets per second.

Both control plane operations had a significant effect on the distribution of timing probe RTT. However, flow installation had a much larger impact. When the controller processed 500 packets per second, the RTT approximately doubled. When the control plane installed a flow rule for each packet, the RTTs doubled at approximately 50 packets per second.

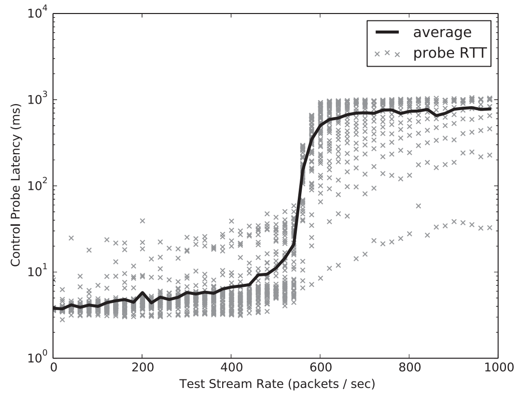

Figure 9: Timing probe RTT as test stream packet rate varies, for test streams that are processed by the control plane.

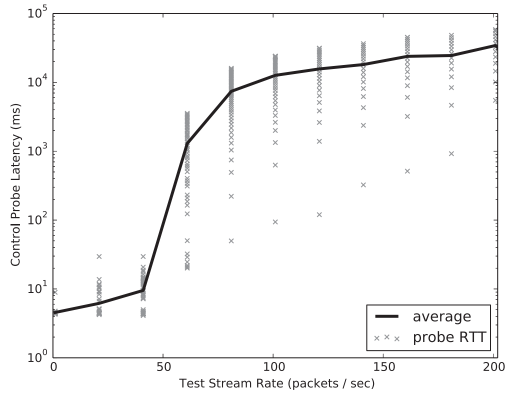

Figure 10: Timing Probe RTT as test stream packet rate varies, for a test stream where each packet invokes a rule installation.

We also observed that the distribution of RTTs changed very quickly when transmitting test flows that placed load on the control plane, with average RTTs increasing in under 1 second.

Control plane bottlenecks. In both experiments, we observed a significant nonlinear jump: from <10ms to >500ms when the stream rate went above approximately 550 packets per second, in the forwarding decision trials; and from <10ms to >1000ms when the stream rate went above approximately 50 packets per second in the flow installation trials. We measured the resource usage of the physical components of the control plane in trials with 5 second long test flows, shown in Table 2, and concluded that the bottleneck was the switch CPU, in agreement with previous benchmarks [14, 35]. We found that the low level software interconnecting the forwarding engine to OpenFlow agent running on the switch was unoptimized and put extremely high load on the switch's CPU: the forwarding engine driver did not support *packet coalescing* or *polling*, as a result each time a packet needed to travel between the forwarding engine and controller, there were many expensive context switch and copy operations.

The impact of background traffic. To measure the effect of background traffic on the timing attack, we ran trials of the timing attack while replaying our background trace at different rates.

	Packet Processing						Flow Installations					
Rate	**10**	**30**	**50**	**70**	**90**	**200**	**10**	**30**	**50**	**70**	**90**	**200**
Switch CPU Usage	11.0%	15.7%	25.0%	26.0%	32.0%	65.0%	5.9%	16.7%	28.7%	60.0%	87.0%	99.0%
Controller CPU Usage	2.6%	6.0%	11.8%	15.0%	19.0%	38.0%	3.3%	10.0%	10.0%	13.0%	10.0%	10.0%
Switch Memory Usage	8.5%	8.5%	8.5%	8.5%	8.5%	8.5%	9.5%	9.7%	9.7%	9.9%	10.0%	10.4%
Controller Memory Usage	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%
Switch to Controller Bandwidth	0.20%	0.65%	1.05%	1.55%	1.85%	4.20%	0.17%	0.70%	0.72%	1.03%	1.40%	0.35%

Table 2: Resource usage of control plane components for packet processing and flow installation at different rates. The switch's CPU is the primary bottleneck, especially for flow installation.

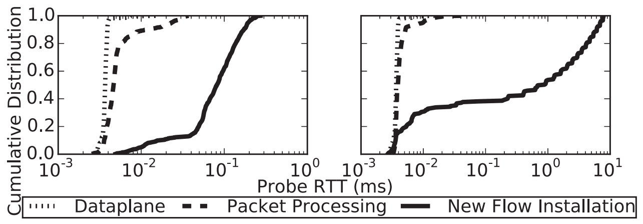

Figure 11: Empirical Timing Probe RTT distributions while sending test streams at 50 and 500 packets per second.

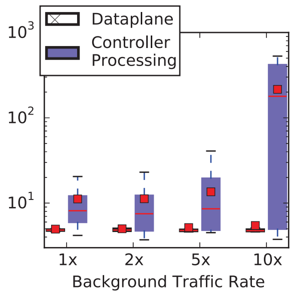

Figure 12: Probe RTT distributions as background traffic varies, for test streams that cause the controller to process 500 packets per second.

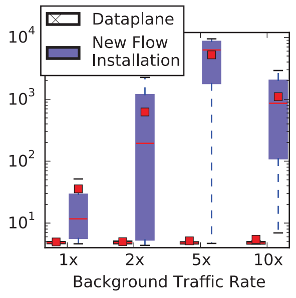

Figure 13: Probe RTT distributions as background traffic varies, for test streams that cause the controller to install 50 flow rules per second.

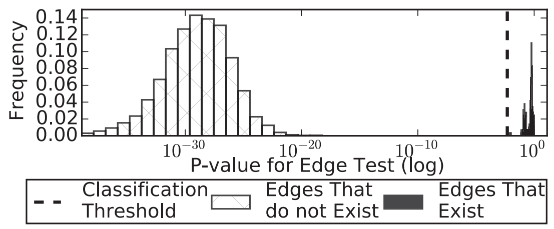

Figure 14: P-values of tests to determine if rules corresponding to edges in the network's communication graph were installed on the switch.

Source IP	Destination IP	Avg Probe RTT
1.1.1.1	1.1.1.5	3.41 ms
1.1.1.2	1.1.1.5	8.08 ms
1.1.1.3	1.1.1.5	8.65 ms
1.1.1.4	1.1.1.5	6.68 ms
1.1.1.6	1.1.1.5	3.70 ms
1.1.1.7	1.1.1.5	3.67 ms
1.1.1.1	1.1.1.6	6.52 ms

Table 3: Inferring the ACL in Table 1. Low RTTs signal to the attacker that the packets were dropped by the switch's ACL table before they reached its MAC learning table, which would forward the packets to the controller, increasing RTT.

As Figures 12 and 13 show, higher rates of background traffic did not weaken the effectiveness of our attack. Conversely, higher background traffic rates actually amplified the effect of the test streams on timing probe RTT – the difference between the RTT distributions during dataplane and control plane processing of test streams increased with background traffic rate, making it *easier* for an adversary to figure out whether the control plane was processing test packets. This effect occurred because timing probes RTTs increased non- linearly with control plane load, and the background traffic placed a small baseline load on it.

We also observed that when the background traffic put a heavier load on the switch, the switch began to drop a large fraction of flow installation requests, causing the reduction in probe RTT depicted in Figure 13 when moving from a 5X to 10X background traffic replay rate. Even in these scenarios, transmitting a test flow still caused a statistically significant shift to timing probe RTTs.

4.4 Attack Implications

As we described in Section 3, the timing attack can reveal many security sensitive details about a network. Here, we evaluate the example attacks from that section.

Learning host communication patterns. We used the timing attack to build a communication graph of the hosts active during a portion of our background trace. Nodes in this graph correspond to hosts, and edges exist between hosts that communicate. There were approximately 150 hosts, each of which communicated with an average of 3.1 other hosts.

Passive monitoring provides knowledge of *which hosts exist on the network* because of broadcast ARP traffic. However, in a layer 2 or 3 network, passive monitoring cannot inform an adversary about which *pairs of hosts communicate*. We used our attack to infer the complete host communication graph (*i.e.*, a graph with a node for each host and an edge between each pair of hosts that communicated) in under an hour. We performed one attack trial to check for the existence of a forwarding rule that corresponded to each edge of the graph. Each trial took 1 second and sent 500 test packets and 10 timing probes into the network. We compared the RTTs of the 10 timing probes with a baseline sample of RTTs, using a t-test. If the p value was lower than 0.05, we considered the distributions different, which indicated that the rule did not exist. Otherwise, we considered the distributions statistically equivalent, which indicated that the rule existed. We observed *no* false positives or negatives with these parameters, because the p-values in tests where no flow existed were orders of magnitude lower, as Figure 14 shows.

Learning ACL entries. To test ACL rule inference, we installed the ACL rules in Table 1 onto the switch. The ACL table matched

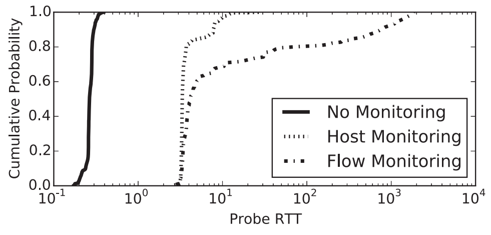

Figure 15: ECDFs of probe RTTs while sending test packets to detect OpenFlow based monitoring.

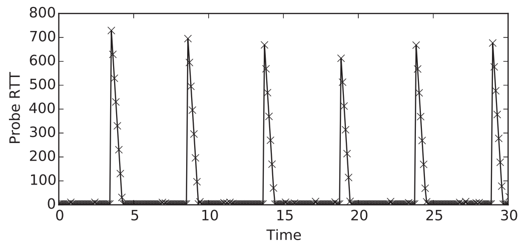

Figure 16: Timing probe RTTs for a control application that polls the switch for statistics updates about 1000 **flow rules every 5 second.**

packets before the forwarding table. The measurements of the RTT in the attack trials is summarized by Table 3.

The trials that correspond to IP address pairs that are allowed by the ACL reach the MAC learning table, which forward it to the controller and increases the timing probe RTTs. The trials that correspond to IP address pairs that are blocked by the ACL do not reach the MAC learning table, lowering the timing probe RTT to baseline values. These observations would enable an attack to learn the entirety of the ACL table.

Learning about monitoring controllers. We implemented two simple monitoring applications on our Ryu controller: a *host monitor* that counts the number of bytes sent by each host; and a *flow monitor* that counts the number of bytes in each flow. Both applications counted by installing per host or per flow rules into a counting table that matched packets before the main forwarding table.

Figure 15 shows the distributions of timing probe RTTs in trials where we sent the all pairs test stream with 32 different IP addresses, as defined in Section 3. We sent the test streams at a rate of 70 packets per second. The timing probe RTTs remained at baseline levels when the controller did not run either monitoring application. When the controller ran the flow monitoring application, we observed a distribution that corresponded to approximately 70 new flow rule installations per second according to Figure 10, which correctly indicated that the control plane was installing approximately 1 rule per test stream packet. When the controller ran the host monitoring application, the timing probe RTT shifted to a distribution that corresponded to approximately $\frac{70}{32}$ new flow rule installations per second, which correctly indicated that the control plane was installing approximately 1 rule every 32 packets, or 1 rule per unique test stream IP address.

Figure 16 shows the probe RTTs that we observed during the 30 seconds that followed our trial with the flow monitor control applications, which polled the switch for flow updates every 5 seconds. The probe RTT spikes clearly indicate this polling interval.

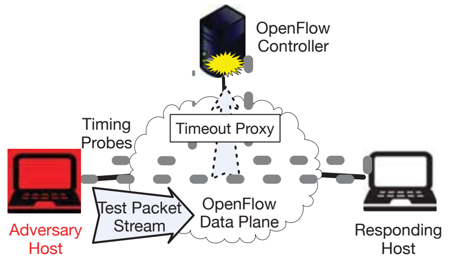

Figure 17: The timeout proxy sends a default packet forwarding instruction to the switch if the controller doesn't respond within a threshold period of time, normalizing the RTT of any potential adversary timing probes.

4.5 Attack Evaluation Summary

In summary, our results demonstrated that the control plane timing attack was highly effective in a testbed with physical OpenFlow equipment and realistic background traffic:

- The attack is highly effective. In trials with test flows with rates of 500 packets per second, we were able to correctly predict what role the control plane played in processing the test flow with $> 99\%$ accuracy.
- Control plane load has a high impact on timing probe RTT: processing only 500 packets per second in the control plane more than doubled timing probe RTT, on average; flow rule installation had approximately 10X greater of an effect. The primary bottlenecks were inefficient software and hardware between the switch's forwarding engine and its OpenFlow software agent that connects to the control server.
- Increased background traffic load increased the difference between RTTs observed when the control plane played different roles, benefiting the attack.
- With repeated attack trials, adversaries could learn which pairs of devices communicated, infer the entries in a switch's ACL, and uncover the monitoring behavior of a controller with nearly perfect accuracy.

5. DEFENDING THE CONTROL PLANE

To the best of our knowledge, there has been no implementation and testing of timing attack defenses for OpenFlow networks. The attack proposed in this paper is challenging to defend against because of its generality: existing defenses can only protect against specific *instantiations* of the attack because they do not address the underlying issue of correlation between *control plane load* and *control plane processing time*. For example, deploying a MAC based access control system [13] to drops packets from MAC addresses that are not pre-authorized can stop the attack if an adversary uses *spoofed ARP timing probes*. However, it does not stop adversaries that use other types of timing probes.

In this section, we describe a *robust* and *general* software based defense to control plane timing attacks that can run on existing physical OpenFlow switches and evaluate it on our testbed.

5.1 An OpenFlow Timeout Proxy

The core idea of our defense, depicted in Figure 17, is to normalize the RTT of attack timing probes to prevent the adversary from learning about control plane load. We implemented this solution using a *timeout proxy* that is interposed between the switch's forwarding engine and the control server.

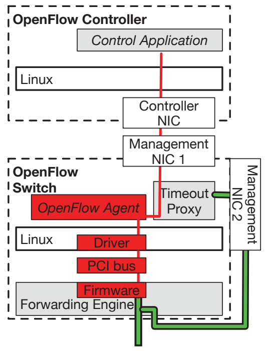

Figure 18: The timeout proxy avoids the standard bottlenecks between the data and control planes, and quickly sends default instructions to the switch when control requests time out.

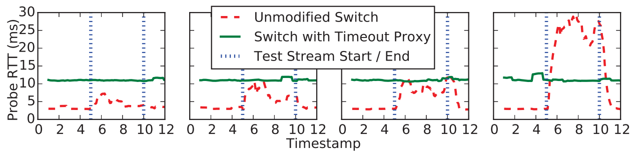

Figure 19: Probe RTTs while sending test packets that are processed by the control plane, with and without the timeout proxy defense, for trials with test packet rates of 400, 600, 800, and 1000 packets per second.

The timeout proxy tracks each packet that the forwarding engine sends to the controller. If the controller does not respond within a threshold period of time, the proxy makes a forwarding decision for a packet by matching it against a table containing *default flow rules* and sends it to the forwarding engine.

Later, if the controller sends a forwarding instruction for the packet, the proxy drops it to avoid duplicate packets. On the other hand, if the controller responds too soon, (*i.e.* before the threshold period of time has passed), the proxy queues the response until the threshold amount of time has elapsed. The proxy does not interfere with any of the other OpenFlow interactions between the switch and the controller, such as flow installation or polling.

The controller installs the default flow rules to the proxy, and sets the timeout threshold that determines how long the proxy waits before considering a request timed-out. Increasing the interval gives the control plane longer to respond, and decreases the number of timeouts that will occur. However, it also increases the round trip time of all packets processed by the control plane. Since the control plane is usually only invoked at flow set up, the threshold can be set quite high without causing a user-noticeable impact. In our experiments, we used a threshold of 11 ms, approximately 3 times the average control plane RTT.

Implementation. Figure 18 shows how the timeout proxy interconnects with the standard components of an OpenFlow switch. The forwarding engine sends a copy of each controller-bound packet to the proxy via the switch's second management interface. This connection avoids all of the standard bottlenecks between the controller and the switch that we discussed in Section 4. The proxy also monitors each packet transmitted between the switch and the controller to determine which requests the controller has responded to. Our implementation encodes forwarding instructions as IP-ToS tags. It maps each possible default action to a unique tag. To send an instruction to the switch for a packet, it places the appropriate tag onto the packet and then sends it to the forwarding engine, where it is matched against a special proxy managed table that takes the appropriate action based on the ToS tag.

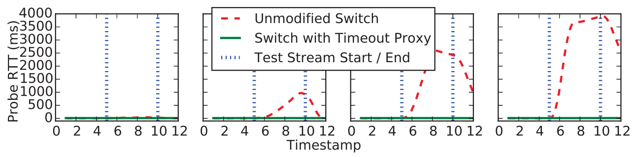

Figure 20: Probe RTTs while sending test packets that cause the control plane to install new rules onto the switch, with and without the timeout proxy defense, for trials with test packet rates of 400, 600, 800, and 1000 packets per second.

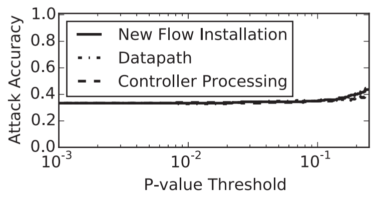

Figure 21: Attack accuracy while running the proxy.

This switch behavior conforms to all OpenFlow specifications, as OpenFlow switches do not guarantee that they will always respect the controller's forwarding instructions. The proxy is implemented as an efficient C++ application with two threads: a *proxy thread* that forwards packets between the switch and controller, and maintains a list of outstanding requests to the control plane; and a *timeout thread* that checks the list for requests that have timed out.

5.2 Defense Evaluation

We evaluated our timeout proxy implementation in the OpenFlow testbed described in Section 4. The timeout proxy ran directly on our Pica 8 3290 switch, configured to send controller responses to the switch within 10 - 11 ms of the original request. We focused on answering three questions: (1) what effect does the timeout proxy have on timing probe RTTs; (2) does the timeout proxy impact an adversary's ability to accurately predict what actions the control plane is taking in response to test packet streams; and (3) what are the upper limits of the timeout proxy, with respect to how many packets per second it can handle?

Timing probe RTT. Figure 19 shows the effect of the timeout proxy on timing probe RTTs while an adversary sends test packets into the network that require control plane processing. Without the proxy, latency was low (3ms) during the pre-attack period when control plane load was low, but drastically increased when the adversary host sent a test stream that increased control plane load. The timeout proxy normalized the control plane delay: during the period of light load, before the test stream was sent, it queued control plane responses for 7ms; during the period of higher load, while the test stream was transmitting, it sent a default action to the switch whenever the control plane did not respond within 11 ms.

Figure 20 plots timing probe RTTs while the adversary host sent test packets into the network that caused the control plane to install a new flow rule for each test stream packet. Each flow installation puts significant load on the control plane, thus without the timeout proxy there was a large increase to probe RTT when the adversary transmitted the test stream. The timeout proxy again normalized the amount of delay added to the probe RTTs, even in extreme load scenarios that would have added >1000 ms of latency to the timing probes.

Impact on timing accuracy. Figure 21 shows our attack's accu-

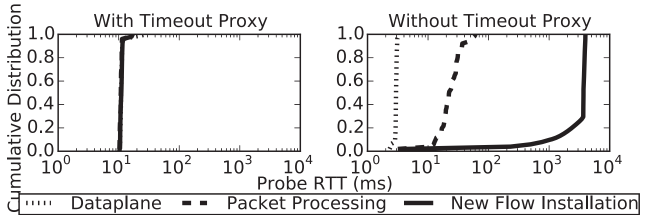

Figure 22: Probe RTT distributions for test streams that invoked the control plane.

Rate	100	500	1000	5000	10000
Without Timeout Proxy					
25th percentile RTT	2.83	3.84	18.78	4276.2	–
50th percentile RTT	2.94	4.56	22.38	4460.1	–
75th percentile RTT	3.25	5.22	29.38	4637.8	–
With Timeout Proxy					
25th percentile RTT	10.83	10.63	10.55	10.84	11.97
50th percentile RTT	11.08	10.86	10.85	11.16	15.05
75th percentile RTT	11.27	11.12	11.17	13.35	20.46

Table 4: timing probe RTT statistics as attack rate changes.

racy in 300 trials where we used the t-test technique described in Section 3 to predict whether a test stream transmitted at a rate of 500 packets per second was *processed by the dataplane*, *processed by the control plane*, or *caused new flow installations*.

While using the proxy, there was *no* p-value threshold that resulted in a high accuracy. This is in contrast to the identical trials that we did without the proxy, where we saw that there was a wide range of p-values that yielded perfect accuracy (Figure 8). The proxy had this impact on the timing attack's accuracy because it kept the distribution of the timing probe RTTs nearly identical, regardless of what role the controller played in processing the test stream packets, as Figure 22, a plot of the average ECDFs of each class of test flow with and without the proxy, shows. The similarity between these timing probe RTT distributions suggests that the proxy would be effective against *any* statistical or machine learning based approach to learn the control plane's role by analyzing timing probe RTT differences.

Upper limits. In our testbed, the current implementation of the timeout proxy was effective against test streams with rates of up to approximately $10,000$ packets per second. Table 5.2 summarizes statistics for probe RTTs while transmitting test streams that caused the control plane to process packets. When using the timeout proxy, a test stream of $5,000$ packets per second shifted the timing probe RTT distribution less than a test stream of 100 packets per second did on an undefended network. The timeout proxy performed even better in trials where test streams caused the control plane to install rules. On an undefended network, we found that a test stream that caused 10 flow rule installation requests per second had as much of an impact on probe RTTs as test streams that caused $5,000$ flow rule installation requests when the timeout proxy was in use.

Test streams with rates above $10,000$ packets per second were a significant strain on several components of our testbed, even with the timing proxy, including both the switch's and controller's CPU. Two simple improvements would provide protection against attacks with much higher rates: moving the timeout proxy application off the switch, to a dedicated server with a stronger CPU; replacing Ryu, the Python based control platform we used in our testbed, with a control platform designed for performance, such as [27, 15, 31].

5.3 Minimizing Network Impact

In this section, we argue that the timeout proxy would not have a large impact on non-attack network traffic and discuss ways that a concerned network operator could configure the proxy to further optimize network performance while still gaining the benefits of increased defense against control plane timing attacks.

Forwarding correctness. Controllers almost always rely on a default action to handle packets that they are unsure of how to forward. These forwarding actions must be correct, in that they cannot break the network (*e.g.* cause packet storms or lose packets), but may not be optimal (*e.g.* they may use more bandwidth than necessary). For example, many MAC learners flood packets with unknown destination addresses, or broadcast them across a pre-computed minimum spanning tree of the network. By configuring the timeout proxy to use the same default actions as the controller, it can provide a similar guarantee that timeout decisions are *correct* but possibly *non-optimal*.

Limiting timeouts. The timeout proxy will only processes a small fraction of traffic, since the control plane is generally only invoked for flow set up, and since most control plane requests will not time out. To further reduce the fraction of these packets that timeout actions are applied to, a network operator can apply filtering, and only allow the timeout proxy to send a default response for classes of packets that *may be timing probes*. For example, in our example attacks we used ARP replies as timing probes; a network operator could configure the proxy to only send default responses for ARP replies, guaranteeing that it would have no impact on all other classes of network traffic.

Passive timing attack detection. The timeout proxy also serves as an ideal platform for a detection-based defense against control plane timing attacks. A network operator could configure the timeout proxy so that it *never* sends a timeout response to the switch, and simply collects logs of all the attempted switch to controller requests. This log could then be analyzed using statistical or heuristic based approaches to detect hosts in the network potentially running the timing attack. In this mode of operation, the timeout proxy would not only provide security benefits, but also likely *improve* the performance of the network, as it actually *increases* the capacity of the channel between the data and the control plane by avoiding common bottlenecks, as Figure 18 depicts.

6. CONCLUSION

Control plane timing attacks allow adversaries to learn about Software-defined networks without needing to compromise their infrastructures. We developed a more refined timing attack that reveals new kinds of sensitive information about an OpenFlow network and demonstrated that it has high accuracy against real OpenFlow hardware. We also proposed and evaluated a robust software based defense, capable of running on existing physical OpenFlow switches. As SDN adoption grows and networks deploy increasingly more advanced control plane applications, it is likely that timing attacks will be observed in the wild. Our work provides a comprehensive first look at their potential, and offers a deployable and effective defense.

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, DARPA grants numbers HR0011-16-C-0061 and PO-0004103, and ONR grant number N00014-15-1-2006.

7. REFERENCES

[1] National collegiate cyber defense competition. http://www.nationalccdc.org.

[2] Nsa uses openflow for tracking... its network. http://www.networkworld.com/article/2937787/sdn/nsa-uses-openflow-for-tracking-its-network.html.

[3] Protected repository for the defense of infrastructure against cyber threats. http://predict.org.

[4] Ryu. http://osrg.github.io/ryu/.

[5] Ryu 1.0 documentation: Router. https://osrg.github.io/ryu-book/en/html/rest_router.html.

[6] P. 8. Pica 8 3290 datasheet. http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf.

[7] A. Andreyev. Introducing data center fabric, the next-generation facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[8] M. Appelman. Performance analysis of openflow hardware. 2012.

[9] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web*, pages 621–628. ACM, 2007.

[10] B. B. Brumley and N. Tuveri. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*, pages 355–371. Springer, 2011.

[11] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM Computer Communication Review*, 37(4):1–12, 2007.

[13] Cisco. Configuring port security. http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/port_sec.pdf.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[15] D. Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[17] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. Sdx: A software defined internet exchange. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 551–562. ACM, 2014.

[18] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[19] W. Hurd. The data breach you haven't heard about. http://www.wsj.com/articles/the-data-breach-you-havent-heard-about-1453853742, 2016.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 3–14. ACM, 2013.

[21] K. Jarvis, J. Milletary, and T. Intelligence. Inside a targeted point-of-sale data breach, 2014.

[22] M. Joye, P. Paillier, and B. Schoenmakers. On second-order differential power analysis. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 293–308. Springer, 2005.

[23] 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.

[24] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

[25] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[26] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. *Computing*, 2:2, 2005.

[27] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[28] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[29] 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.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[31] B. S. Networks. Project floodlight. http://www.projectfloodlight.org/floodlight/.

[32] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[33] 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.

[34] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *NDSS*, 2013.

[35] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Enabling practical software-defined networking security applications with ofx. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[36] Wikipedia. Student's t-test — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Student's%20t-test&oldid=723048300.

[37] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.