Monitoring LAN Traffic Flows via Side-Channel Analysis in Software-Defined Networking Switches

Gorka Irazoqui, Berk Gulmezoglu, Mehmet Sinan İnci
Craig A. Shue, Thomas Eisenbarth and Berk Sunar
Worcester Polytechnic Institute
{girazoki, bgulmezoglu, msinci, cshue, teisenbarth, sunar}@wpi.edu

ABSTRACT

Software-defined networking (SDN) provides powerful control over local area network (LAN) traffic. Through OpenFlow [1], a popular SDN protocol, a network controller can provide a series of rules that are locally cached in the data plane of virtual or physical OpenFlow switches. Using these cached rules, the switches direct traffic towards the destination. However, the use of caches on OpenFlow switches may create a side-channel vulnerability. An adversary on the LAN may be able to issue carefully-designed probe packets to confirm whether traffic is occurring on the network or not. In this paper, we examine the efficacy of such side-channel attacks and find that we are able to determine whether traffic has used a given flow rule or not, allowing us to confirm the network usage patterns of other users.

1. INTRODUCTION

Software-defined networking (SDN) is an approach that allows network operators to separate the control plane functions of a network, such as routing, from the data plane forwarding functions. The OpenFlow protocol [22], a popular mechanism for implementing SDN, treats network switches as high-speed data plane forwarding caches. When a packet is received by an OpenFlow switch, it checks to see if it has an existing cached rule that matches the packet. If so, it uses that rule and forwards the packet as the rule dictates. Otherwise, it invokes an "elevation" process in which it asks a logically-centralized system, the OpenFlow controller, for instructions. This caching and elevation process creates side-channel leakage that adversaries can analyze to deduce information about the network.

Recent work has shown that sharing physical resources, most prominently caches, causes unintended information leakage and allows adversaries to recover sensitive data [23, 15, 28, 31, 5, 16, 10, 14]. Other researchers explored side-channel analysis to learn the size of switch flow tables [19] and the OpenFlow rule structure [8].

In this work, we explore the side-channel leakage of shared SDN switches. In particular, we use cache timings to determine if parties were communicating on a network, to whom, and for how long. With such information, an adversary could distinguish regular and honey pot systems [29], identify key network assets, and profile and spy upon other network users, even across subnets and VLANs.

We ask the following research questions: *1) To what extent can adversaries use host-based network measurements to determine whether a network flow rule is being used by others? 2) What role do OpenFlow implementation details, such as timeouts and cache table hierarchies, play in enabling traffic confirmation attacks? 3) What factors affect the precision of these attacks?*

In our exploration, we make the following contributions:

Characterization of Cache Side-Channel Potential: We analyzed both physical switches and hypervisor-based virtual switches and found that an adversary can easily distinguish between cached hits and cache misses (which require elevation to an OpenFlow controller). Further, even in the worst case scenario where a virtual switch on a hypervisor is used with a high-performance controller running on a high-end server, the adversary still succeeds.

Characterization of SDN Switch Parameters: We perform a full characterization of the OpenFlow switch's flow table size, hard and soft timeout values, and rule implementation approaches. This information enables an adversary to perform more effective profiling of traffic.

Confirmation of Traffic via End-Host Probes: We have successfully created a process allowing an adversary on a LAN to confirm traffic destined to a victim machine. Further, when multiple OpenFlow switches are used, we allow colluding adversaries to localize the origin of communication to the victim machine. The adversary can use the side-channel analysis to determine the destination and, roughly, the source of the communication along with the time and approximate duration of the communication.

2. BACKGROUND AND RELATED WORK

Our background and related work can be grouped into three broad categories: SDNs, cache-based side-channel attacks, and attacks on SDNs. We now describe each in detail.

2.1 SDNs and OpenFlow

OpenFlow can be used with both physical switches and

virtual ones, such as the connectors found in VM hypervisors. Open vSwitch (OVS) [26] is an open-source virtual switch and is one of the most popular implementations of the OpenFlow protocol. OVS is divided into a kernel module, which provides data plane processing for each packet, and a user space daemon, called `ovs-vswitchd`, which communicates with the OpenFlow controller and updates the kernel data structures with rules from the controller.

2.2 Cache Attacks and Side-Channel Analysis

Timing-based side-channels were studied as early as 1996 by Kocher [18] and further expanded by Page [25]. These works studied the feasibility of timing attacks to extract information from cryptographic libraries. It was later shown that similar side-channels are caused by the presence of caches [24, 6], as shown in the Flush and Reload attack on RSA by Yarom *et al.* [30] or Gruss *et al.* [11]. Later, the Prime and Probe on the last-level cache was proposed by Irazoqui *et al.* [14] and Liu *et al.* [21]. Moreover, in 2015, Inci *et al.* [13] demonstrated the Prime and Probe technique in the Amazon Elastic Cloud (EC2) while Lipp attacked existing Android smartphones [20], showing the practicality of side-channel attacks in deployed systems. While such cache techniques have been investigated in a number of scenarios, to our knowledge, they have not been investigated as a potential source of information extraction in SDN networks.

2.3 Attacks on SDNs

SDN security has been explored in numerous ways. Kloti *et al.* [17] applied the STRIDE threat analysis methodology [12] to the OpenFlow protocol and evaluated the feasibility of DDoS attacks and information disclosure. On the defensive side, Braga *et al.* [7] create a method using traffic flow analysis to detect DDoS attacks. Similarly, Ballarad *et al.* [4] use OpenSAFE to control the traffic routing using network monitoring devices whereas Shin *et al.* [27] use CloudWatcher to ensure that all necessary packets are checked by a security monitor to prevent attacks. In SPHINX [9], the authors create a real time detection system for attacks on SDN that profiles network behaviors and alerts on deviations.

The amount of work combining SDNs and side-channel analysis is more limited. Acs *et al.* [3] demonstrated the feasibility of cache attacks in Named Data Networking (NDN) routers and the ability of the adversary to learn whether a nearby user requested the same content. Leng *et al.* [19] showed how an attacker could learn the cache size and exploit overflows while Cui *et al.* [8] showed that an adversary could learn about the structure of rules used in the network.

Our work differs in that we are able to learn traffic flow patterns in the network, determining which hosts are popular on the network and the group of interested clients.

3. MEASUREMENT METHODOLOGY

Given Open vSwitch's (OVS) popularity, both in hypervisors and in physical switch implementations, such as OpenWrt [2], we examine the potential for side-channel attacks on OVS implementations. Using a lab environment, we test OVS in a basic SDN network and induce cache contention to determine the timing of cache hits and misses.

3.1 OVS Implementation Details

OVS maintains two types of caches: a *microflow* cache and a *megaflow* cache. The microflow cache allows high-performance lookups of heavily used flows, but is restricted to exact matches (i.e., the matching criteria in the rules may not specify wildcards for any entries). The megaflow cache is a secondary, larger cache that can store entries containing wildcards, masking off the source address, for example, when the rule only depends on the destination address. Combined, these caches can dramatically increase the speed in which packets are processed in OVS.

One of the main distinctions between the microflow and megaflow tables is the type of rule they store. Entries in the microflow cache must be fully specified and thus match only a single flow. These are called *fine-grained* rules. However, the megaflow table entries can be *coarse-grained* rules, meaning they can have wildcards in some fields. For example, using the 5-tuple network notation (IP_{src}, IP_{dst}, $Port_{src}$, $Port_{dst}$, transport protocol), a flow entry of type (*, 1.2.3.4, *, 80, TCP) could only be stored in the megaflow cache table since it uses wildcards for the source IP address. While coarse-grained rules cannot be stored in the microflow cache, an instantiation of the coarse-grained rule could be stored in the microflow cache. For example, if a packet with network tuple of (5.6.7.8, 1.2.3.4, 50147, 80, TCP) came along, OVS may choose to cache that exact network tuple in the microflow cache while keeping the coarse-grained rule in the megaflow cache. This approach allows OVS to use the faster microflow table for actively used flows while supporting the flexibility of course grained rules.

When ordering a switch to cache a rule, the OpenFlow controller will often specify a timeout to indicate that the rule should be pruned after a period of time, regardless of whether the flow tables are full or not. A *soft timeout* indicates that a rule should be removed if it has gone unused for the specified duration. A *hard timeout* indicates that, regardless of whether the rule has been used or not, it should be removed after a specified duration. An adversary can use these timeout values to detect pruned flow rules.

3.2 Probing Strategy

We will issue probes from an end-host in the network, denoted as the adversary, to determine whether a rule is cached in the switch. We will essentially measure the round-trip time (RTT) between our adversary and a response from another host on the network to determine whether the appropriate flow entry was cached in the switch between the two hosts or not. Since the switch must query a controller when a cache miss occurs, we expect a slower round-trip on the first packet in a flow where a cache miss occurs.

Likewise, if a coarse-grained rule is in use, we expect the second RTT to be smaller than the first, but slower than the third. The change in RTT between the second and third exchanges are due to the performance difference in the megaflow cache (used in the second exchange) and the microflow cache (used in the third exchange). The second exchange will trigger OVS to cache a fine-grained flow instance of the coarse-grained rule. The fourth exchange, and all subsequent exchanges, should have the same performance properties of the third exchange.

In our first set of measurements, we examine the timing of the two scenarios: 1) a microflow or megaflow cache hit and 2) a microflow and megaflow cache miss. In particular, we want to determine if the probing attacker can reliably distinguish between the two cases.

In the second set of measurements, the adversary will use our ability to distinguish cache hits and misses to learn more about the settings and policy at the OpenFlow controller, along with the types of rules it caches at the network switches. This preliminary information helps the adversary craft an ideal set of attacks. We use the following steps:

1. Characterize Timeouts: We determine what soft or hard timeout values a switch uses for flow rules, if any.

2. Create and Test Diverse Flows: We vary the header parameters of a set of network packets to trigger different types of OpenFlow flow rules. For example, we create packets where only the destination IP address is varied. The first packet caches a new rule at the switch. We measure the RTT in subsequent packets to determine if the RTT is consistent with a cached or uncached packet. This allows an adversary to learn which parameters are used in the rules and whether those rules are fine-grained or coarse-grained.

3. Determine Flow Table Capacity: The adversary can determine the size of the flow table by sending packets that will cause new rules to be installed. Once the flow table is filled, old, unused flows will be removed. The adversary can detect this by measuring the RTT of packets matching those old rules and seeing a delay indicating a cache miss.

With a third set of measurements, the adversary can launch traffic confirmation attacks. If the controller uses solely fine-grained rules, the adversary has limited ability to learn about other users' traffic since the adversary will never share a rule with other users and thus experience caching differences based on other user's behavior. Likewise, if a controller uses largely static, coarse-grained rules, the adversary will learn only limited information since the traffic will be highly aggregated. Therefore, the adversary learns the most in networks that have rules that are between these extremes: rules that are coarse enough for the adversary to share rules with others, but fine enough to limit aggregation to allow the adversary to profile a small group of users.

The adversary selects a victim host or group of hosts it wishes to profile. The attacker then creates packets that match the flow rules created by the victim's traffic and measures the RTT to determine whether the rules were cached or not hence indicating victim access.

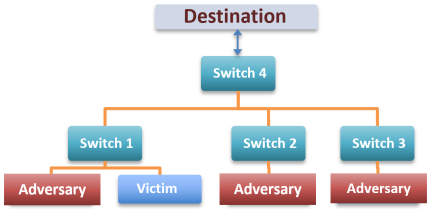

Figure 1: Cascaded switch setup

The adversary may also desire to know traffic patterns over a period of time to learn how the traffic changes. If the adversary wanted to learn whether a victim used a particular flow rule during a given interval, additional measurements are required. To launch such a longitudinal study, the adversary needs a mechanism to ensure that unused flow rules are periodically purged, allowing the adversary to then test whether the rule was re-installed (and thus actively used by the victim) or remains unused.

The adversary has two features that it can use to detect unused flows. The first is simple timeout values the controller may have applied to the cached entry in the switch. If the switch purges the unused rule after t seconds, the adversary can perform a cache check after t seconds to see if it is still in cache. The other approach is an active measure: the adversary simply uses the knowledge it previously learned about the flow table size and creates a set of packets that will cause rules that fill the flow table, pushing out the other rules. The adversary can then later check if the victim's flow rule is in the cache, which would indicate that the victim used the flow rule after the adversary cleared the cache.

3.3 The Role of Multiple Switch Networks

When multiple OpenFlow switches are employed in a network, the adversary may be able to use the caching behavior of the switches to localize its victim.

The OpenFlow controller has options about how it installs flow rules into switches. It can simply install flows on-demand: when a switch asks about a packet, the controller tells that switch, and only that switch, how to forward the packet. Alternatively, the controller could determine the path between the source host and destination and inform each switch on the path about the flow rule that should be cached. If an adversary performs measurements about how the controller operates, it can use these caching behaviors to determine which portion of the network contains the victim.

For example, consider a network as shown in Figure 1. In this case, the adversary controls three hosts, each connected to a different switch. If the adversary wishes to determine the location of a victim is that is talking to a particular destination, and such communication would result in a coarse-grained rule specific to that destination, the adversary can perform measurements on each switch to determine whether

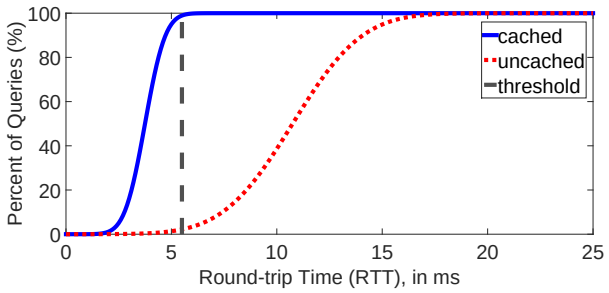

Figure 2: RTTs on an Archer c7 (500 trials per series). The blue series shows cached results and the dotted red series indicates a cache miss. The threshold (the dashed vertical line) is used to distinguish both distributions.

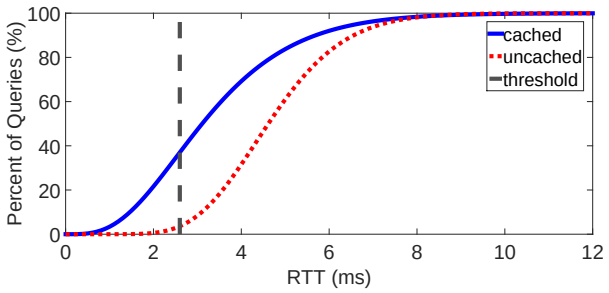

Figure 3: RTTs for a KVM hypervisor scenario (500 trials per series). The blue series shows cached results and the dotted red series indicates a cache miss. The selected threshold (dashed line) minimizes false positives.

the rule is cached. If the adversary performs a request via Switch 1, the rule will become cached in both Switch 1 and Switch 4. If the adversary then probes via Switch 2, the adversary can determine if the victim has asked for the rule via Switch 2 (the RTT indicates a cache hit on Switch 2) or not (the RTT indicates a cache miss). Using this approach, the adversary can localize the victim machines.

4. EVALUATION AND RESULTS

We now describe our experimental setup for our measurements and the results that we obtained from the experiments.

4.1 Experiment Setup

We use multiple components in our experiments. The first is a TP-LINK Archer c7 router that we flashed with custom OpenWRT firmware with the Open vSwitch kernel module installed. This switch receives instruction from an OpenFlow controller running on a server with 8-core Intel Xeon CPU and 32 GB of memory. We performed experiments using both the Python-based POX and the C++-based NOX controllers. For the other hosts, we used notebook PCs.

4.2 Distinguishing Cached Flow Entries

We begin by exploring whether the adversary can distinguish between the two caching scenarios: 1) a megaflow cache hit and 2) a megaflow cache miss. To do so, we create flows that will allow us to test these scenarios and measure the RTT for each. In our tests, we create 500 new, distinct flows. Upon elevation, the NOX controller pushes a coarse-grained rule to the switch. Accordingly, the first packet in the flow is a cache miss for both the microflow and megaflow caches, the second packet is a microflow cache miss and a megaflow cache hit. At this point, the switch creates a custom fine-grained rule for the flow and stores in in the microflow cache, causing the third packet in the flow to return from the microflow cache. For brevity, we exclude microflow analysis from our experiments.

In Figure 2, we show the results of this experiment. We can see that both megaflow cache hit and miss scenarios are distinct and have little overlap, indicating that an adversary can indeed distinguish the types of cached entries in practice. The results from the POX controller were similar, except that the cache miss scenario took longer (over 25ms), making it even easier for the adversary to distinguish. Since NOX is the more conservative for showing attack potential (i.e., it is the most challenging case for the adversary), we only present NOX results due to space constraints. In our subsequent results, we focus only on distinguishing between cache hits and cache misses (elevations to the controller).

We then examine the adversary's worst case scenario, a hypervisor-based OVS implementation. In Figure 3, we show the RTT distribution of 500 cached and uncached probes between two VMs hosted on a KVM hypervisor with a dual core CPU using OVS as the virtual switch. The controller is the same 8 core CPU used in the previous experiment. While the distributions are closer, an adversary could select a 2.5ms threshold to minimize false positives while providing a 40% detection probability. Even in the worst case, the adversary gains a useful side-channel.

4.3 Determining Rule Timeout Values

The OpenFlow controller can set both soft timeout and hard timeout values. Both of these can be valuable signals for an adversary to confirm if a flow is being actively used.

In our first test, we first explored detection of a soft timeout value. With this goal in mind, we sent a packet that caused a new flow to be created. We then waited for a variable amount of time and issued a probe that matched the same flow. If the RTT associated with the probe matched our "uncached" time, we knew that the flow's soft timeout had elapsed before the probe was issued. In order to successfully determine a soft timeout value of t seconds, the attacker has to send $t+2$ flows in $(t+1)(t+2)/2$ seconds. In Figure 4, we show the results of 200 trials for this experiment. The actual timeout was configured to be 10 seconds. All results in which the probe was within 10 seconds had short RTTs, matching the cached behavior in Figure 2 while all probes delayed by at least 10 seconds had a longer RTT, matching the uncached results. This approach allows the adversary to discover the controller's soft timeout value.

We next try to determine the existence of any hard timeout

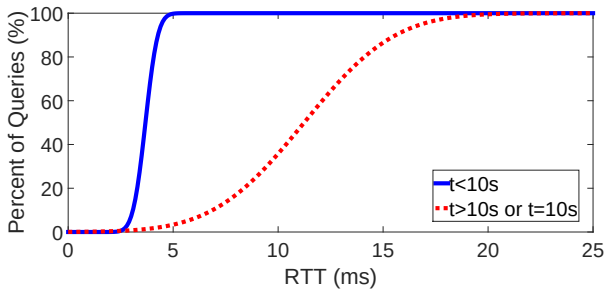

Figure 4: RTT distributions for all queries less than the timeout (dashed line) and all queries greater than the timeout value (dotted line). The RTT grew significantly after the timeout.

value. In this experiment, we create a new flow and then continually issue probes for it every second. This repeated probing continually renews the flow to avoid a soft timeout. For a rule with t seconds hard timeout, this process would only take $t + 1$ seconds and $t + 1$ probes. When the RTT of the probes grows, reflecting a cache miss, the attacker discovers the hard timeout value. Our accumulated results for 200 different trials are similar to those in the soft timeout experiment: RTTs are low before the timeout occurs, but high afterwards. We omit the plot of these results for brevity.

From these experiments, we see that an adversary can easily learn a controller's soft and hard timeout values by measuring RTTs. The hard timeout values are not affected by other traffic, while soft timeout values may be affected.

4.4 Characterizing Flow Rule Parameters

The adversary needs to know whether the controller installs fine-grained or coarse-grained rules in the switches. For coarse-grained rules, the adversary must discover which elements in the rule are wildcarded and which are static.

To discover the types of rules in use, we perform experiments where we vary a single element of the packet headers to determine when cache misses occur. If varying a field causes a cache miss, the adversary learns that the varied field is not wildcarded (otherwise, the cached rule would have matched). For example, if an adversary wanted to determine if the source port is static in rules, the adversary would create multiple flows that have identical network and transport layer header fields, with the exclusion of a varying source port, and then evaluate the RTT for the responses. A long RTT response may indicate an uncached rule.

In our experiments, we create scenarios in which we vary the following fields independently: source MAC, source IP, source transport layer port, destination IP, destination port.

Figure 5 shows the results of 200 these probes for each header field tested using both fine-grained rules and coarse-grained rules where we independently varied each of the source MAC address, destination IP address, and source IP address header fields. We see that when using fine-grained rules, the RTTs for each were higher, indicating cache misses. When using coarse-grained rules, the RTTs are lower. However, due to underlying network mechanisms

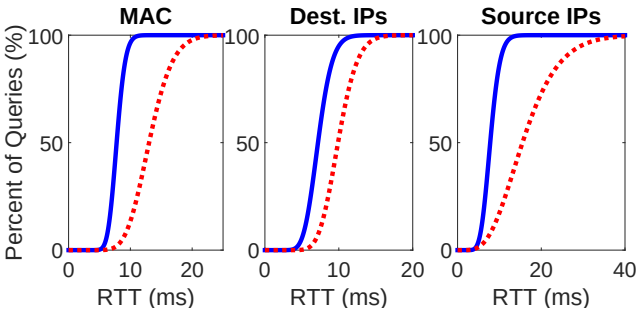

Figure 5: When varying the targeted field with coarse-grained rules (the left, black series), the low RTTs indicate that the same rule matched multiple probes. However, with fine-grained rules (the right, blue series), the higher RTTs indicate that each new flow created a cache miss.

(including the need to ARP for new MAC addresses and whether flow rules are installed symmetrically), the results do not perfectly match cached behavior when coarse-grained rules are used. Nevertheless, an adversary can distinguish the distributions and determine whether coarse-grained or fine-grained rules are in use. The results for transport layer ports are even more distinct and easier for an adversary to distinguish, but we omit these results for brevity.

4.5 Determining the Flow Table Capacity

When the controller does not set timeout values for its rules, the adversary must manually flush all flows by filling the table with junk flows. To do so effectively, the adversary must infer the size of the table by generating different packets and by checking whether a specific flow has been evicted or not from the flow table. In order to test this approach, we generated a packet that would install the first cache rule, r_0. We then created n additional packets that would cause the switch to cache n junk rules $(r_1, r_2, \ldots, r_n)$. We then measure the RTT when sending another packet matching rule r_0. If the RTT indicates the record is cached, then the cache table can hold at least n entries. Otherwise, we likely know the table can hold n entries or fewer[1].

Figure 6 shows the results of 200 trials in which probes were issued before and after the 1000 junk flows were stored. We see low RTTs when we install rules less than the table size. However, once the table size is exceeded, the RTTs jump to values that indicate a cache miss, showing the flow table size was exceeded and the earliest entry was evicted. We see similar behavior when the table is limited to 100 entries. OVS appears to use a least-recently used eviction policy since even if the accumulated entries exceed the flow table size, a cache miss was not observed unless n number of flows were installed between probes.

4.6 Traffic Monitoring Attack Effectiveness

[1]Unrelated traffic may cache other rules, filling the table before n entries are stored. The adversary can adjust for this since the table capacity is often a power of 2 or 10.

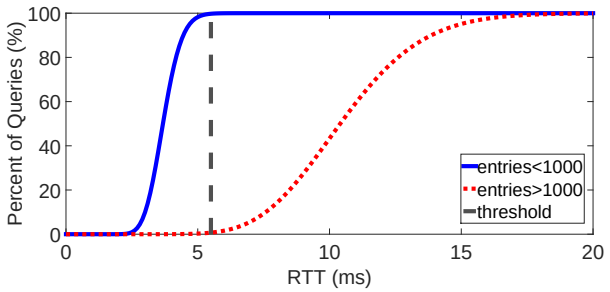

Figure 6: Across 200 trials, the RTT was low when the 1000 entry table had not been filled but high when over 1000 entries were stored, indicating the table flushing approach is effective.

		Detected As	
		Cached	Uncached
Flow Status	Cached	89.47%	10.52%
	Uncached	1.16%	98.83%

Table 1: Confusion matrix of hard timeout evictions using a detection threshold of 5.5 ms

The adversary has two choices to provide longitudinal traffic confirmation attacks: either use one of the soft or hard timeout values or use a rule eviction set that fills the entire flow table. The first is stealthier but also limits the rate at which traffic confirmation attacks are launched. The latter is more robust, but it is easier for a defender to detect.

Eviction using the timeout values: This approach consists of monitoring whether the targeted flow exists in the cache after t number of seconds, with t representing the hard timeout value. If the flow exists in the cache, we deduce that a potential victim installed it. We did not choose the soft timeout because it involves higher number of false negatives. Table 1 represents the results obtained by emulating a victim that randomly decides whether it installs the targeted flow among 200 timeslots with a 30 second hard timeout value. The attacker probes the targeted flow once every 30 seconds. The distinguisher threshold is set to 5.5ms following the distributions observed in Figure 2. It can be observed that although the number of false negatives is almost 5 times higher than the false positives, and that in both cases we obtain a success rate minimum of 90%.

Eviction using the eviction set: The second approach consists of ensuring the eviction of a flow from the table by filling it with the attackers own data. We again emulate a victim making random accesses to a server at 200 different timeslots with a 1000 entry flow table switch. The results are presented in Table 2. This time, the number of false positives is higher than the false negatives, rising to 13.19%.

4.7 Impact of Multiple Switches

We now test whether the attacker is able to infer the distance (in terms of number of switch hops) between itself and the victim. We utilize a single controller that communicates with three different switches reactively. The number of switches having cache misses essentially acted as a multiplicative factor for the RTT delay associated with a cache miss. We use the hard timeout value approach using a 30 second timeout due to its better performance with false positives. The attacker randomly accesses a server from one of the three switches. Table 3 summarizes the results obtained using the threshold from in Figure 2. The adversary can correctly distinguish when the victim is on the same switch or 1 hop away with over a 85% success rate. However, we found that for the cases of a victim being 2 hops away, the adversary had only a 63% success rate. Finally, the case where no access was made was only clearly distinguishable in 52% of the cases. The adversary can better detect the victim's location when the adversary is nearby.

		Detected As	
		Cached	Uncached
Flow Status	Cached	94.50%	5.50%
	Uncached	13.19%	86.81%

Table 2: Confusion matrix of eviction set evictions using a detection threshold of 5.5 ms

		Detected As			
		Same switch	1 hop	2 hops	No access
Victim Accesss	Same switch	88.0%	8.0%	4.0%	0.0%
	1 hop	0.0%	88.83%	11.43%	0.0%
	2 hops	0.0%	20.57%	62.5%	16.67%
	No acccess	0.0%	0.0%	48.39%	51.61%

Table 3: Confusion matrix of the flow rule detection with the thresholds of 5.5, 8.5 and 11.5 ms for each hop distance using the hard timeout eviction mechanism.

5. IMPLICATIONS AND DISCUSSION

Our study shows side-channel analysis is viable in both physical and virtual switch OpenFlow networks. An adversary can easily distinguish between cached and uncached entries to learn network configuration details, such as key rule parameters, timeout values, and the switch table capacity. The adversary can localize victims by exploiting switch hierarchies and cached entries and confirm whether the victims are sending certain traffic or not.

Network operators can use our results to help defend their networks. By monitoring the elevation traffic at their OpenFlow controllers, they can detect any repeated querying that could be associated with an adversary trying to confirm traffic. Further, rapid increases in elevation requests could be a symptom of an adversary trying to fill switch caches. Finally, defenders should use the longest timeouts on rules as feasible since short timeouts provide adversaries with a convenient mechanism to determine if a flow is no longer in use.

6. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation, under grants 1318919, 1314770 and 1422180.

7. REFERENCES

[1] OpenFlow Switch Specification 1.4.0. https://goo.gl/IL3KFv, 2014.

[2] About OpenWrt. https://wiki.openwrt.org/about/start, 2016.

[3] Acs, G., Conti, M., Gasti, P., Ghali, C., and Tsudik, G. Cache privacy in named-data networking. In *ICDCS 2013*.

[4] Ballard, J. R., Rae, I., and Akella, A. Extensible and scalable network monitoring using opensafe. *Proc. INM/WREN* (2010).

[5] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In *CHES* (2014), pp. 75–92.

[6] Bernstein, D. J. Cache-timing attacks on AES. Tech. rep., 2005.

[7] Braga, R., Mota, E., and Passito, A. Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *LCN 2010*.

[8] Cui, H., Karame, G. O., Klaedtke, F., and Bifulco, R. Fingerprinting software-defined networks. *arXiv preprint arXiv:1512.06585* (2015).

[9] Dhawan, M., Poddar, R., Mahajan, K., and Mann, V. SPHINX: Detecting security attacks in Software-Defined Networks. In *NDSS 2015*.

[10] Fangfei Liu and Yuval Yarom and Qian Ge and Gernot Heiser and Ruby B. Lee. Last level cache side channel attacks are practical. In *S&P 2015*.

[11] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security 15*.

[12] Hernan, S., Lambert, S., Ostwald, T., and Shostack, A. Threat modeling-uncover security design flaws using the stride approach. *MSDN Magazine-Louisville* (2006), 68–75.

[13] İnci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Tech. rep., IACR Cryptology ePrint Archive, 2015.

[14] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES. In *S&P 2015*.

[15] Irazoqui, G., İnci, M. S., Eisenbarth, T., and Sunar, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies Symposium 2015*.

[16] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.

[17] Kloti, R., Kotronis, V., and Smith, P. OpenFlow: A security analysis. In *ICNP, 2013* (2013), IEEE, pp. 1–6.

[18] Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*.

[19] Leng, J., Zhou, Y., Zhang, J., and Hu, C. 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).

[20] Lipp, M., Gruss, D., Spreitzer, R., and Mangard, S. Armageddon: Last-level cache attacks on mobile devices. *CoRR abs/1511.04897* (2015).

[21] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *S&P 2015*.

[22] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. Openflow: Enabling innovation in campus networks. *SIGCOMM 2008*.

[23] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The Spy in the Sandbox - Practical Cache Attacks in Javascript. *CoRR abs/1502.07373* (2015).

[24] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: the case of AES. In *CT-RSA 2006*.

[25] Page, D. Theoretical use of cache memory as a cryptanalytic side-channel, 2002.

[26] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. The Design and Implementation of Open vSwitch. In *12th USENIX NSDI* (2015).

[27] Shin, S., and Gu, G. Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). In *ICNP 2012*.

[28] Suzaki, K., Iijima, K., Toshiki, Y., and Artho, C. Implementation of a memory disclosure attack on memory deduplication of virtual machines. *Fundamentals of Electronics, Communications and Computer Sciences* (2013).

[29] Wang, P., Wu, L., Cunningham, R., and Zou, C. C. Honeypot detection in advanced botnet attacks. *International Journal of Information and Computer Security 4*, 1 (2010), 30–51.

[30] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security 14*, pp. 719–732.

[31] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS 2012*.