2017 IEEE 37th International Conference on Distributed Computing Systems

Flow Reconnaissance via Timing Attacks on SDN Switches

Sheng Liu
University of North Carolina
Chapel Hill, NC, USA
Email: shengliu@cs.unc.edu

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA
Email: reiter@cs.unc.edu

Vyas Sekar
Carnegie Mellon University
Pittsburgh, PA, USA
Email: vsekar@andrew.cmu.edu

***Abstract*—When encountering a packet for which it has no matching forwarding rule, a software-defined networking (SDN) switch requests an appropriate rule from its controller; this request delays the routing of the flow until the controller responds. We show that this delay gives rise to a timing side channel in which an attacker can test for the recent occurrence of a target flow by judiciously probing the switch with forged flows and using the delays they encounter to discern whether covering rules were previously installed in the switch. We develop a Markov model of an SDN switch to permit the attacker to select the best probe (or probes) to infer whether a target flow has recently occurred. Our model captures practical challenges related to rule evictions to make room for other rules; rule timeouts due to inactivity; the presence of multiple rules that apply to overlapping sets of flows; and rule priorities. We show that our model enables detection of target flows with considerable accuracy in many cases.**

I. Introduction

The move toward software-defined networking (SDN) is one of the dominant trends in the networking landscape today. As realized using OpenFlow [3], an SDN *switch* is managed by a *controller* that produces and delivers rules to the switch to handle packets. For our purposes here, each rule simply expresses the network interface to which a flow — i.e., packets sharing the same source and destination addresses and ports — should be directed. When a switch receives a packet for which it has no matching forwarding rule, the switch forwards it to the controller, which will then install a rule for this flow in the switch. Rules delivered to the switch are cached at the switch until one of two events occurs [3]: either the switch's rule cache fills up and so some rules need to be *evicted* (typically using a shortest-time-remaining policy [2]) to make room for other rules arriving from the controller, or else a rule simply *expires* due to not being matched by a flow for the rule's timeout duration.

The delay suffered by a flow that matches no rule in the switch and so is forwarded to the controller is much larger than the delay suffered by a flow that matches an already cached rule. Like other caches that support eviction, expiration, or both, it has been noticed that an SDN switch therefore enables a timing side-channel attack whereby an attacker can submit a *probe* flow (potentially forging the source address) to a switch and timing the delay it suffers through the switch [9], [17], [20], [21]. This delay can be measured, for example, by observing the flow packets emitted from the switch or by measuring the delay until receiving the response. In this way, the attacker can determine if a rule covering the probe flow was already installed in the switch. Previous works (see Section II) have used this observation to detect the use of SDN in a network [17] or to reverse-engineer packet-forwarding logic [9], [20], [21].

In this paper, we introduce a different opportunity to use this side channel, specifically to perform reconnaissance on the activity in the network. That is, the timing side channel described above that reveals the presence or absence of a rule covering a flow in a given switch can leak information about what parties communicated recently. In the context of certain applications, this leakage could be used as a form of reconnaissance. For example, the attacker could use this timing attack to detect whether an intrusion-detection system (IDS) logged a record to a logging site (as some enterprise defenses are configured to do, e.g., [23]), which might, in turn, reveal whether the IDS detected an activity that the attacker recently attempted.

Inferring information from this side channel is complicated by the fact that rules can *overlap* — i.e., a flow can be covered by multiple rules deployed to the switch. In this case, the switch matches the flow to the highest *priority* rule that covers it, where the priority is an attribute of the rule that the controller conveys to the switch when installing the rule. This feature clouds the results of the aforementioned timing channel, since even if the attacker learns that a rule covering its probe flow was already installed in the switch, it might not know which specific rule was matched. Consequently, calculating the best probes to make, and what can be inferred from them, is a challenge for the attacker.

In this paper we construct a Markov model of the switch that an attacker can use to determine the probe(s) to best infer whether a target flow occurred recently in the network. This model permits the adversary to estimate the probability distribution over the possible rule cache states, in the face of rule expirations and evictions. The challenge in constructing this model is that the number of possible cache states is massive, and so we construct our model incrementally: we first build a high-fidelity model that can scale to only a few rules, but then construct a more approximate model that can scale substantially better. We then show how the adversary could use this model to select probe flows to issue to the switch, to

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.281

infer the most that it can about whether a target flow occurred recently.

To make our model(s) concrete, we detail the attacker's calculations under the assumption that each flow's occurrence is Poisson distributed. We allow the attacker knowledge of estimates of the Poisson parameters for flows in the network, as well as knowledge of the rule set from which the controller selects rules to deploy to the switch. This rule set might be exposed to the attacker through previous compromises in the network and might even be released voluntarily (e.g., as Stanford University has [13]). Similarly, the Poisson parameters could be inferred through previous compromises of flow logs or simply through knowledge of the roles of various machines in the network.

Using Mininet [1] and the Ryu controller [4], we illustrate that our model can substantially improve flow-level reconnaissance via timing side channels over what the attacker could accomplish with naive attacks alone (i.e., using the target flow itself as the probe). In the context of rule sets and flow rates for which a side-channel-based detector is possible, our model improves the accuracy of these attacks in our tests by only $\sim 2\%$ on average. However, for certain subclasses of rule sets and flow rates, this improvement can grow to 23% or more, yielding an average accuracy approaching 85%, when naive attacks barely reach 62% accuracy. And, because our model permits the attacker to choose the best probe flows to attempt from among those probes that are viable for him, our model enables the attacker to accomplish near-optimal inference even when using sub-optimal (but still possible) probes.

To summarize, our contributions are as follows:

- We introduce flow reconnaissance attacks using timing side channels arising from the reactive operation of SDN switches. To our knowledge, this paper is the first attempt to leverage this timing side channel to perform flow reconnaissance in an SDN network.
- We construct a Markov model of an SDN switch to estimate the switch-rule distribution, and propose a method by which the adversary can use this distribution to select an optimal probe. Our model captures complexities related to rule eviction, expiration, and priorities.
- Through experiments on Mininet, we demonstrate that our proposed model can enable reconnaissance attacks that, in some cases, are substantially more accurate than naive flow reconnaissance attacks.

II. Related work

Timing attacks on SDN networks of the type that motivates our work were first considered by Shin and Gu [17], who proposed fingerprinting network forwarding logic by observing different response times of exchanged packets. Cui et al. [9] implemented this attack on a real-world testbed and presented a countermeasure by delaying the first few packets of a flow. Sonchack et al. [20] demonstrated a more sophisticated inference attack that an adversary can use to time the controller even if the injected packets do not induce a reply message. John et al. [21] extended the work of Sonchack et al. and explored attacks to infer host communication patterns, ACL entries, and network monitoring. However, none of these works take into account the complexities related to rule eviction and expiration, along with overlapping rules with various priorities. Compared with these, our paper introduces the opportunity to leverage this timing attack specifically for flow reconnaissance and proposes a Markov model to capture these complexities.

Several papers (e.g., [7], [12]) model hit rates of timeout-based network caches. In contrast to these models, our model deals with not only timeout events and cache evictions, but also rule overlaps and priorities. Considering rule overlaps results in a substantially more complex model.

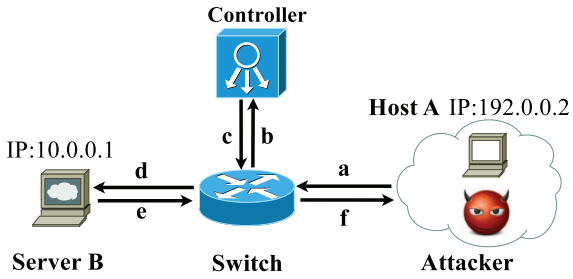
More distantly related to our work are various works studying other security problems impacting SDN or solutions building from SDN (e.g., [10], [11], [16], [18], [19]). For instance, a denial-of-service attack on the control plane, and a corresponding mitigation method, were proposed by Shin et al. [19]. Hong et al. [11] introduced a network topology poisoning attack to hijack network connections by exploiting SDN vulnerabilities. Other than exploiting SDN vulnerabilities, many works enhance SDN security or leverage SDN to provide more convenient security solutions. For example, FRESCO [18] is an SDN security framework to facilitate the modular composition of OpenFlow-enabled security modules. SPHINX [10] is a framework to detect attacks based on network flow graphs. Porras et al. [16] presented a security extension of the control layer to detect and arbitrate conflicts due to flow rule installations from multiple applications.

III. Attack Overview

Let us recall the rule setup procedure of an SDN network. When a host sends a flow to the SDN network (step a in Figure 1), the switch will check whether there is a covering rule in the flow table and, if so, performs the corresponding action (step d). If not, the switch forwards the flow to the controller (step b, c) to query for a new rule setup. This flow setup delays the first packet of the flow. The adversary could use this observable delay (specifically, the delay to receive f) to not only determine whether a rule covering the flow already existed in the switch table, but also to detect specific other flows based on the relationships between rules and flows.

A rule in the flow table covers addresses and ports in the IP header, and has a priority for matching precedence, a set of associated actions, and a countdown timer for the rule that stores the amount of time before it expires. If a rule is in the switch, then a flow must have been matched to it within the preceding timeout duration for that rule, as otherwise the rule would have expired.

To conduct this side-channel attack, the attacker must be able to estimate the delay suffered by its probe packets. There are several ways to do so (e.g., [9], [20]). One simple method is to gain a presence on the hosts from which it wants to launch probes, since it can then easily measure the durations until it receives the corresponding responses. Or, if the attacker can compromise a host, it can listen for a response packet to a



Match Fields	Action	Timeout	Priority
10.0.0.0 to 192.0.0.255	Forward to port 2	10 s	3
...	...	...	...
152.0.0.0 to 202.0.0.255	Forward to port 3	5 s	2

Fig. 1: Example attack

probe that it forges as coming from another host connected to the same switch port. To time probes which do not evoke a reply or that the attacker cannot otherwise measure, the adversary can leverage other techniques (e.g., [6], [20]). For example, the attacker can simultaneously inject such probe flows alongside unforged flows into the network, chosen to match no rules already in the switch (and so to be forwarded to the controller). By noticing whether the processing of the unforged flows by the controller interferes with the probe flows (e.g., the responses to the unforged flows are delayed), the adversary can determine if the switch already had a rule covering the probe flows. In this paper, we are agnostic to the specific method that the attacker uses to estimate the delays seen by probe packets and simply assume the attacker can do so reliably.

A. Example Attack

One simple attack is that the attacker sends a flow to the switch with a spoofed packet header (i.e., changing the source IP address to a victim's address). As illustrated in Figure 1, if the attacker would like to probe whether host A has visited some web server B, he can just deliver two HTTP request flows, one with its own IP address as the source (denoted f_1) and the other one with source address that of host A (denoted f_2). If the attacker has not communicated with server B before, then the time to receive the response (message f) to flow f_1 is $t_{\text{fetch}} + t_{\text{setup}}$, where t_{fetch} is the response time if a covering rule already exists in the switch and t_{setup} is the delay for rule initialization (including the time required for the communication between the switch and the controller, the controller processing time, and the time of flow entry insertion into the switch flow table).

In contrast, the response time for f_2 varies according to whether host A has visited B recently, and thus the value can be either t_{fetch} or $t_{\text{fetch}} + t_{\text{setup}}$. As t_{setup} is not negligible, by comparing the response time for f_2 to that for f_1, the attacker can determine whether the rule covering f_2 existed in the switch or not. Since each rule has a timeout policy [3], the

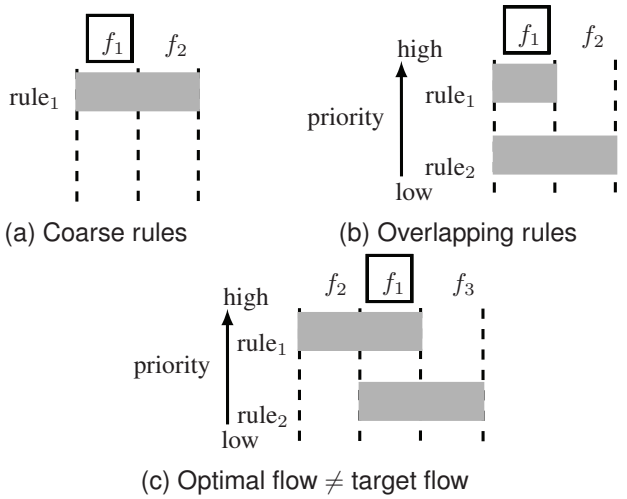


Fig. 2: Challenges arising from rule overlap, discussed in Section III-B

presence of the rule covering f_2 in the switch indicates that A visited B during its timeout interval.[1]

This attack might be used for reconnaissance in many scenarios. For example, by probing the message exchange between a user's host and a particular web server and so learning whether this user has visited the server recently, the attacker might infer private information about the user. Similarly, the attacker could use this attack to probe whether an intrusion-detection system (IDS) logged a detection record to a logging database (as some enterprises configure their malware detectors to do [23]), i.e., by probing for the logging flow between the IDS system and the database. The result might allow the attacker to infer whether the IDS detected an activity the attacker attempted and so the detection capabilities or policies of the IDS.

B. Challenges

Although this attack seems straightforward, a successful attack needs to overcome several challenges. Here, we discuss several factors that the attacker needs to consider.

1) Rule granularity: If the controller uses fine-grained microflow rules (for example, each rule covers only one flow [8]),[2] the attacker can easily tell which flow occurred by inferring which rule exists in the switch. However, for coarser rules that leverage wildcards to cover multiple flows, it is harder for the attacker to identify the occurrence of the target flow by purely recognizing the existence of a corresponding rule, since the presence of other covered flows besides the target can cause false detections. For example (Figure 2a), if rule$_1$ covers both f_1 and f_2, it is not easy to differentiate the occurrence of target flow f_1 from f_2 by just sending f_1 as a probe.

[1]OpenFlow defines two kinds of timeouts, hard timeout and idle timeout. The former causes the rule to be removed after exactly the given number of seconds, while the latter expires the rule when it has matched no packets in the given number of seconds.

[2]A microflow specifies the exact 32-bit IP addresses and 16-bit ports without any wildcard.

2) Rule Overlap: Another complicating factor is rules that overlap. OpenFlow [3] allows overlapping rules (i.e., two rules that cover the same flow) with different priorities to coexist in the switch. Upon arrival of a flow, the flow is matched to the highest priority rule in the switch, if any. This further clouds the results of this timing side-channel attack, since even if the probe matches a rule in the switch, the attacker may not know exactly which rule was matched. Note that, if a miss happens, the covering rule with the highest priority will be installed in the switch.

However, by carefully selecting multiple probes, the adversary may be able to reduce this ambiguity. For example (Figure 2b), consider two rules rule_1 and rule_2, where rule_1 covers f_1, and rule_2 covers f_1 and f_2. Suppose that rule_1 has higher priority than rule_2. If in an effort to detect the occurrence of f_1, the attacker just sends f_1 and measures a response time indicating that a rule covering f_1 is in the switch, then the adversary still cannot be certain which rule (rule_1 or rule_2) was matched. Matching rule_2 could be caused by the previous arrival of flow f_2. But, if the adversary sends both f_1 and f_2 and observes that f_1 matched but f_2 missed, then he can confirm the existence of rule rule_1 and absence of rule_2, therefore inferring the previous occurrence of f_1.

In fact, the optimal probe may not always be the target flow. For example (Figure 2c), assume rule_1 covers f_1 and f_2, and rule_2 covers f_1 and f_3, with rule_1 having higher priority than rule_2. If the adversary would like to probe the presence of f_1, then the optimal choice is f_2 and not f_1: the match of flow f_2 would guarantee the existence of rule_1 (which can only be installed by f_1 and f_2), while the match of flow f_1 will only indicate at least one of these two rules is in the switch (which can be caused by any of these three flows).

Such rule dependencies introduce new sources of complexity that make our attack difficult to reason about. Accounting for these effects and searching for the optimal probes is one of the central contributions of the model that we develop in this paper.

3) Limited Cache Size: SDN switches have a limited flow table size and can only cache a limited number of rules, which further complicates our attack. For example, if the target flow occurred but the corresponding rule was evicted by other rules, then the attacker may get a false negative result. The adversary thus needs an approach that accounts for the likelihood of such false negatives.

These challenges require the attacker to construct a model to describe the switch state by capturing complexities related to rule evictions, rule expiration and rule dependencies, and also an approach to optimally choose probes based on the model.

C. Threat Model

The goal of the adversary is to infer the presence of private communications between computers on a network. We assume the attacker has a presence on one or more computers in the same network, and that it can use these hosts to initiate flows on the network and observe responses to them (or otherwise measure the delays they suffer). That is, we suppose that the attacker has a set of flows he can use to launch the timing attack. We assume that the adversary knows the network policies (i.e., flow rules) and what flows they cover. We think this assumption is reasonable; e.g., some networks have published their network policies online (e.g., [13], [22]) or the attacker can obtain them through previous compromises or other reverse engineering techniques (e.g., [15]). Moreover, we allow the attacker to have prior knowledge about the distribution of flow occurrence (see Section IV-A1 for more details) and the size of the switch flow entry table (which the attacker could obtain from the specification of the switch hardware or through previous attacks [14]).

IV. Modeling an SDN Switch

In order to detect the occurrence of the target flow, the attacker should first try to construct the state of the switch flow table using crafted probes. However, as mentioned in Section III-B, many factors (e.g. rule dependency, rule expiration, and eviction) make it difficult to predict the exact state of the switch. Therefore, here we devise a model of a switch that permits him to compute a distribution over the possible switch states.

The policy enforced by an SDN switch is expressed by a set Rules of *rules*. Each rule prescribes an action to be performed on the flows in a set of flow identifiers (IP header 5-tuples) that it covers. Since here we are not concerned with the action prescribed by a rule, we simply consider each rule $\mathsf{rule} \in \mathsf{Rules}$ to be the set of flow identifiers it covers; we therefore write $f \in \mathsf{rule}$ to denote the fact that rule covers flow f.

Multiple rules are allowed to cover a given flow. That is, it is possible that $\mathsf{rule}_j \cap \mathsf{rule}_{j'} \neq \emptyset$ for distinct rules rule_j and $\mathsf{rule}_{j'}$; these rules are said to *overlap*. Rules are therefore ordered by priority, such that rules that overlap cannot have the same priority. When the controller is informed of a flow f, it responds with the highest priority rule that covers f. We write $\mathsf{rule}_j > \mathsf{rule}_{j'}$ to indicate that rule rule_j is higher priority than rule $\mathsf{rule}_{j'}$. The relation $>$ is a total order.

A rule delivered to a switch is cached at the switch until it is evicted or expires. The contents of a switch cache are represented by an element of $((\mathsf{Rules} \times \mathbb{N}) \cup \bot)^n$. That is, the cache $\mathsf{cache}[1 \ldots n]$ is an array of length n, the i-th element of which is a rule/integer pair ($\mathsf{cache}[i] = (\mathsf{rule}, \mathsf{exp}) \in \mathsf{Rules} \times \mathbb{N}$) or undefined ($\mathsf{cache}[i] = \bot$). exp is the time remaining before rule expires. We require that $\mathsf{cache}[i] = \bot$ implies that $\mathsf{cache}[i+1] = \bot$ if $i < n$, and that if $\mathsf{cache}[i] = (\mathsf{rule}_j, *)$ and $\mathsf{cache}[i'] = (\mathsf{rule}_j, *)$, then $i = i'$. In other words, a rule can occupy only one location in the cache. In an abuse of notation, we will sometimes write "$\mathsf{rule}_j \in \mathsf{cache}$" as a shorthand for "$\mathsf{cache}[i] = (\mathsf{rule}_j, \mathsf{exp}_j)$ for some i and some $\mathsf{exp}_j \in \mathbb{N}$".

A. Basic model

In this section we describe a simple Markov chain that describes the evolution of the contents of an SDN switch cache. This model captures the effects of idle timeouts, hard timeouts, and evictions. In this Markov chain, a *state* is simply

a switch cache contents as described above. We denote the ℓ-th state by cache_ℓ.

The transitions out of state cache_ℓ represent the possible events that might occur during the passage of a fixed duration Δ since entering state cache_ℓ. Δ is selected so that the probability of multiple flows arriving in Δ time is negligible. To compute Δ (and these transition probabilities), we assume knowledge of the probability p_f that a flow f arrives in a window of size Δ. Our model assumes that flow arrival events are independent, even for the same flow identifier f.

The types of state transitions in our model are as follows:

- **Flow arrival with covering rule in cache**: If a flow f arrives at the switch for which a covering rule is in cache_ℓ, then the model transitions to $\mathsf{cache}_{\ell'}$ in which the covering rule has been moved to the front of the cache and all timeouts have been adjusted appropriately. Specifically $\mathsf{cache}_{\ell'}$ satisfies the following properties, where rule_j is the highest priority rule in cache_ℓ that matches f and is located at index i in cache_ℓ (i.e., $\mathsf{cache}_\ell[i] = (\mathsf{rule}_j, \mathsf{exp}_j)$) and where t_j denotes the timeout for rule rule_j:
  - $\mathsf{cache}_{\ell'}[1] = (\mathsf{rule}_j, \mathsf{exp}_j - 1)$ if rule_j has a hard timeout and $\mathsf{cache}_{\ell'}[1] = (\mathsf{rule}_j, t_j)$ if rule_j has an idle timeout;
  - for $i' = 1 \ldots i-1$, if $\mathsf{cache}_\ell[i'] = (\mathsf{rule}_{j'}, \mathsf{exp}_{j'})$ then $\mathsf{cache}_{\ell'}[i'+1] = (\mathsf{rule}_{j'}, \mathsf{exp}_{j'} - 1)$; and
  - for $i' = i+1 \ldots n$, if $\mathsf{cache}_\ell[i'] = (\mathsf{rule}_{j'}, \mathsf{exp}_{j'})$ then $\mathsf{cache}_{\ell'}[i'] = (\mathsf{rule}_{j'}, \mathsf{exp}_{j'} - 1)$ else $\mathsf{cache}_{\ell'}[i'] = \bot$.
- **Flow arrival with no covering rule in cache**: If a flow f arrives at the switch for which no covering rule is in cache_ℓ, then the model transitions to $\mathsf{cache}_{\ell'}$ in which a covering rule has been brought into cache at the first position; if the cache cache_ℓ was at capacity, then the rule with the smallest remaining time is evicted and all timeouts on remaining rules have been adjusted appropriately. Specifically $\mathsf{cache}_{\ell'}$ satisfies the following properties, where rule_j is the highest priority rule in Rules that covers f and index i is 1 if cache_ℓ was not at capacity or else is the index of the rule $\mathsf{rule}_{j'}$ in cache_ℓ with the smallest remaining time $\mathsf{exp}_{j'}$ among all rules in cache_ℓ:
  - $\mathsf{cache}_{\ell'}[1] = (\mathsf{rule}_j, t_j)$;
  - for $i' = 1 \ldots i-1$, if $\mathsf{cache}_\ell[i'] = (\mathsf{rule}_{j''}, \mathsf{exp}_{j''})$ then $\mathsf{cache}_{\ell'}[i'+1] = (\mathsf{rule}_{j''}, \mathsf{exp}_{j''} - 1)$ else $\mathsf{cache}_{\ell'}[i'+1] = \bot$.
  - for $i' = i+1 \ldots n$, if $\mathsf{cache}_\ell[i'] = (\mathsf{rule}_{j''}, \mathsf{exp}_{j''})$ then $\mathsf{cache}_{\ell'}[i'] = (\mathsf{rule}_{j''}, \mathsf{exp}_{j''} - 1)$ else $\mathsf{cache}_{\ell'}[i'] = \bot$.
- **Timeout**: If the timeout value for a rule is zero, then the rule is removed from the cache. Specifically, let i be the largest value such that $\mathsf{cache}_\ell[i] = (*, 0)$. Then $\mathsf{cache}_{\ell'}$ satisfies: for $i' = i+1 \ldots n$, $\mathsf{cache}_{\ell'}[i'-1] = \mathsf{cache}_\ell[i']$, and $\mathsf{cache}_{\ell'}[n] = \bot$.

Figure 3 shows a simple example of our basic model with three rules $\mathsf{rule}_1, \mathsf{rule}_2, \mathsf{rule}_3$. rule_1 covers f_1; rule_2 covers both f_1 and f_2 (and so rule_1 and rule_2 overlap); and rule_3 covers f_3. As we can see, if no flow occurs, the clock of all the rules

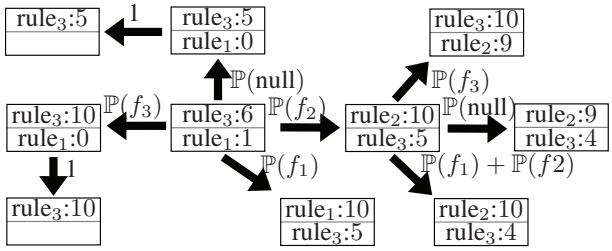


Fig. 3: Basic model example; in each state, shortest remaining time rule is at the bottom; *null* means no flow occurred

in the state will be reduced by one (e.g. $[(\mathsf{rule}_3 : 6), (\mathsf{rule}_1 : 1)]$ to $[(\mathsf{rule}_3 : 5), (\mathsf{rule}_1 : 0)]$). Then, if the counter is zero, the rule expires with probability one (e.g., $[(\mathsf{rule}_3 : 5), (\mathsf{rule}_1 : 0)]$ to $[(\mathsf{rule}_3 : 4)]$). The occurrence of f_1 or f_2 will transition the state $[(\mathsf{rule}_2 : 10), (\mathsf{rule}_3 : 5)]$ to $[(\mathsf{rule}_2 : 10), (\mathsf{rule}_3 : 4)]$, resetting the clock for rule_2 (to its timeout $t_2 = 10$) while the clock for rule_3 is reduced by one. Moreover, if f_2 arrives and rule_2 is not in the cache (e.g. $[(\mathsf{rule}_3 : 6), (\mathsf{rule}_1 : 1)]$), rule_2 will be installed in the cache and rule_1 evicted, since it has the shortest remaining time. Next, we will elaborate how to compute the probability of each state transition.

1) Transition probabilities for Poisson arrivals: The timeout rule takes priority, in the sense that any state cache_ℓ from which a timeout transition is possible (i.e., for which $\mathsf{cache}_\ell[i] = (*, 0)$ for some i) allows *only* that transition (with probability 1). For the other two types of transitions, the method of computing their probabilities depends on the arrival distributions of flows.

Here we give an example of how to compute the corresponding transition probabilities for a particular model of flow arrivals, namely one in which the arrival of each flow (identifier) f is governed by a Poisson process with rate λ_f. We permit the attacker to know λ_f for every flow f; more realistically, the attacker might only be able to estimate λ_f from a known rate λ_j of covering rule rule_j, e.g., by setting $\lambda_f = \lambda_j / |\mathsf{rule}_j|$.

For state cache_ℓ and rule $\mathsf{rule}_j \in \mathsf{Rules}$, let the *relevant flow identifiers for* rule_j, denoted $\mathsf{flowIds}_\ell(j)$, be the set

$$\mathsf{rule}_j \setminus \bigcup_{\substack{\mathsf{rule}_{j'} \in \mathsf{cache}_\ell \\ \mathsf{rule}_{j'} > \mathsf{rule}_j}} \mathsf{rule}_{j'}$$

if $\mathsf{rule}_j \in \mathsf{cache}_\ell$, and

$$\mathsf{rule}_j \setminus \left(\left(\bigcup_{\mathsf{rule}_{j'} \in \mathsf{cache}_\ell} \mathsf{rule}_{j'} \right) \cup \left(\bigcup_{\substack{\mathsf{rule}_{j'} \notin \mathsf{cache}_\ell \\ \mathsf{rule}_{j'} > \mathsf{rule}_j}} \mathsf{rule}_{j'} \right) \right)$$

otherwise. These flow identifiers are "relevant" for rule_j in that the other flow identifiers in rule_j are superceded by other rules in cache_ℓ or, in the case of $\mathsf{rule}_j \notin \mathsf{cache}_\ell$, higher priority rules not in cache. As such, it makes sense to define the rate

$$\gamma_{\ell,j} = \sum_{f \in \mathsf{flowIds}_\ell(j)} \lambda_f \Delta$$

as the *effective* rate for rule rule_j when the cache state is cache_ℓ. Similarly, we define

$$\Gamma_{\ell,j} = \sum_{f \notin \mathsf{flowIds}_\ell(j)} \lambda_f \Delta$$

to be the rate at which flows irrelevant to rule_j in state cache_ℓ arrive. The transition probability from cache_ℓ due to the arrival of a flow matching rule rule_j is then set to

$$\left(\gamma_{\ell,j} e^{-\gamma_{\ell,j}}\right) e^{-\Gamma_{\ell,j}}$$

since one flow relevant for rule rule_j arrives with probability the first factor and no flows irrelevant to rule_j arrive with probability the second factor.

Recall that a Markov chain requires that the probabilities of transitions leaving each state must sum to one. As such, once the above probabilities are computed for each $\mathsf{rule}_j \in \mathsf{Rules}$, they are normalized to sum to one.

2) Scalability: As we will see, the primary challenge in using this Markov chain computationally is the number of states. Specifically, the number of states in this model is

$$\sum_{\substack{\mathsf{Rules}' \subseteq \mathsf{Rules} \\ |\mathsf{Rules}'| \leq n}} \left(|\mathsf{Rules}'|! \cdot \prod_{\mathsf{rule}_j \in \mathsf{Rules}'} (t_j + 1) \right)$$

For example, if $|\mathsf{Rules}| = 10$, each rule has a timeout of $t_j = 100$ steps, and the cache capacity is $n = 8$, then the number of states is about 5.9×10^7.

B. Compact model

In an effort to reduce the number of states, here we describe a more compact but approximate model. In this new model, each state $\mathsf{compact}_\ell$ represents the subset of Rules (of size at most n) that are presently cached, and as such, there are a total of

$$\sum_{n'=1}^{n} \binom{|\mathsf{Rules}|}{n'}$$

states. In doing so, we eliminate from each state any information to implement rule timeouts or evictions. In the remainder of this section, we describe how we compute probabilities to model transitions between states that represent the timeout or eviction of a rule from the cache.

Central to our method of estimating these probabilities is a method of estimating the probability of a given sequence of most recent matches to the rules in $\mathsf{compact}_\ell$. In particular, consider an injective function $u : \mathbb{N} \to \mathbb{N}$ such that u is defined (i.e., $u(j) \neq \perp$) if and only if $\mathsf{rule}_j \in \mathsf{compact}_\ell$. For any such u and $\mathsf{rule}_j \in \mathsf{compact}_\ell$, we interpret $u(j)$ as the number of steps since rule_j was most recently matched (i.e., rule_j was most recently matched at step $\ell - u(j)$).

To assign a probability to the most-recent-match sequence represented by u, it is necessary to define the relevant flow identifiers for rule_j a bit differently than in the previous section, specifically to account for the temporal sequence of these most-recent matches. To do so, we define the relevant flow identifiers for rule $\mathsf{rule}_j \in \mathsf{Rules}$ at step $\ell - k$ to be

$$\mathsf{flowIds}_{\ell,u}(j,k) = \mathsf{rule}_j \setminus \bigcup_{\substack{\mathsf{rule}_{j'} \in \mathsf{compact}_\ell \\ \mathsf{rule}_{j'} > \mathsf{rule}_j \\ u(j') > k}} \mathsf{rule}_{j'} \tag{1}$$

Intuitively, the flow identifiers irrelevant for rule $\mathsf{rule}_j \in \mathsf{Rules}$ at step $\ell - k$ are those covered by higher-priority rules that were not matched in step $\ell - k$. This yields an effective arrival rate of flow identifiers relevant for rule $\mathsf{rule}_j \in \mathsf{Rules}$ at step $\ell - k$ of

$$\gamma_{\ell,u}(j,k) = \sum_{f \in \mathsf{flowIds}_{\ell,u}(j,k)} \lambda_f \Delta$$

Then, for a fixed u, we define the probability of u, denoted $\mathbb{P}(u)$, as

$$\prod_{\mathsf{rule}_j \in \mathsf{compact}_\ell} \left(\gamma_{\ell,u}(j,u(j)) e^{-\gamma_{\ell,u}(j,u(j))} \prod_{k=1}^{u(j)-1} e^{-\gamma_{\ell,u}(j,k)} \right) \times \prod_{\mathsf{rule}_j \notin \mathsf{compact}_\ell} \prod_{k=1}^{t_j} e^{-\gamma_{\ell,u}(j,k)}$$

if $|\mathsf{compact}_\ell| < n$, and

$$\prod_{\mathsf{rule}_j \in \mathsf{compact}_\ell} \left(\gamma_{\ell,u}(j,u(j)) e^{-\gamma_{\ell,u}(j,u(j))} \prod_{k=1}^{u(j)-1} e^{-\gamma_{\ell,u}(j,k)} \right) \times \prod_{\mathsf{rule}_j \notin \mathsf{compact}_\ell} \prod_{k=1}^{u_{\max}(j)} e^{-\gamma_{\ell,u}(j,k)}$$

otherwise, where $u_{\max}(j) = t_j - \min_{\mathsf{rule}_{j'} \in \mathsf{compact}_\ell} (t_{j'} - u(j'))$. In each case, the top factor estimates the probability that a flow relevant to each $\mathsf{rule}_j \in \mathsf{compact}_\ell$ last arrived $u(j)$ steps ago, and the bottom factor estimates the probability that no flow relevant to other rules arrived.

This estimation comes with two caveats, however. First, the inclusion of $\mathsf{rule}_{j'} > \mathsf{rule}_j$ as a restriction in the union on the right side of (1) is correct only if $\mathsf{rule}_{j'}$ was already in the cache before being matched $u(j')$ steps ago; otherwise, this restriction should be removed. Second, we ignore rules $\mathsf{rule}_{j'} \in \mathsf{compact}_\ell$ for which $u(j') < u(j)$, and rules in $\mathsf{Rules} \setminus \mathsf{compact}_\ell$ they may have evicted, since we do not know whether they were also matched before $u(j)$ steps ago. Both of these guesses are shortcomings of using a memoryless (Markov) model.

1) Evictions: When in a state $\mathsf{compact}_\ell$ where $|\mathsf{compact}_\ell| = n$, if a message is received that is not covered by any $\mathsf{rule}_j \in \mathsf{compact}_\ell$, then some rule must be evicted to make room for the rule that the controller sends to the switch. As shown in Figure 4, if current state is $\{\mathsf{rule}_1, \mathsf{rule}_2, \mathsf{rule}_3\}$ and rule_4 is installed (due to the occurrence of f_4), any of the rules in the state may be evicted, leading to three possible state transitions with each rule being exchanged by the new rule. For example, the case

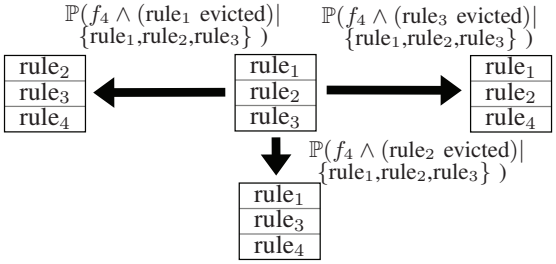


Fig. 4: Eviction example

in which rule_1 is evicted is that the remaining time of rule_1 (its initial expiration time t_1 minus the time since it was last matched, assuming it uses an idle timeout) is smaller than the ones of all the other rules.

In general, we estimate the probability of evicting rule_j conditioned on $\mathsf{rule}_j \in \mathsf{compact}_\ell$ as the ratio of two values: the probability of the joint event $\mathsf{rule}_j \in \mathsf{compact}_\ell$ and rule_j has the smallest remaining time in $\mathsf{compact}_\ell$, and the probability that $\mathsf{rule}_j \in \mathsf{compact}_\ell$. The latter simply corresponds to placing the condition

$$u(j) \leq t_j \tag{2}$$

on u where t_j is the total steps for rule_j expiration, since otherwise, rule_j would have timed out. That is,

$$\mathbb{P}\left(\mathsf{rule}_j \in \mathsf{compact}_\ell\right) = \sum_{u \text{ satisfying (2)}} \mathbb{P}\left(u\right) \tag{3}$$

The probability of the joint event $\mathsf{rule}_j \in \mathsf{compact}_\ell$ and rule_j has the shortest remaining time in $\mathsf{compact}_\ell$ can be estimated by additionally requiring that for any $\mathsf{rule}_{j'} \in \mathsf{compact}_\ell$,

$$t_j - u(j) \leq t_{j'} - u(j') \tag{4}$$

That is,

$$\mathbb{P}\begin{pmatrix}\mathsf{rule}_j \in \mathsf{compact}_\ell \wedge \\ \mathsf{rule}_j \text{ has the shortest} \\ \text{remaining time}\end{pmatrix} = \sum_{u \text{ satisfying (2) and (4)}} \mathbb{P}\left(u\right) \tag{5}$$

Then,

$$\mathbb{P}\begin{pmatrix}\mathsf{rule}_j \text{ has the shortest} \\ \text{remaining time}\end{pmatrix} \Big| \mathsf{rule}_j \in \mathsf{compact}_\ell \Big) = \frac{\text{Eqn. (5)}}{\text{Eqn. (3)}}$$

2) Timeouts: Consider the example in Figure 5, where the current state is $\{\mathsf{rule}_1, \mathsf{rule}_2\}$. If no flow occurs, then each of the rules in the state may expire, leading to two possible state transitions (rule_1 expiration or rule_2 expiration). Specifically, the case for only rule_1 expiring indicates: rule_1 is not matched (or evicted) for t_1 steps, whereas rule_2 is matched within the past t_2 steps.

The probability of rule_j timing out from $\mathsf{compact}_\ell$, given that $\mathsf{rule}_j \in \mathsf{compact}_\ell$, is computed similarly to how the probability of its eviction was computed. The main difference is that the numerator is now the probability of the joint event $\mathsf{rule}_j \in \mathsf{compact}_\ell$ and rule_j has hit its timeout of t_j steps. To estimate this probability, we restrict our attention to functions u satisfying

$$u(j) = t_j \tag{6}$$

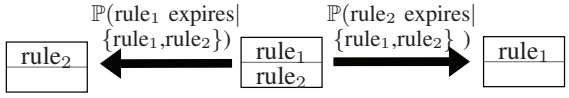


Fig. 5: Expiration example

That is,

$$\mathbb{P}\begin{pmatrix}\mathsf{rule}_j \in \mathsf{compact}_\ell \wedge \\ \mathsf{rule}_j \text{ should time out}\end{pmatrix} = \sum_{u \text{ satisfying (2) and (6)}} \mathbb{P}\left(u\right) \tag{7}$$

and, of course, since (6) implies (2), the condition on the summation can be simplified accordingly. Then,

$$\mathbb{P}\Big(\mathsf{rule}_j \text{ should time out} \;\Big|\; \mathsf{rule}_j \in \mathsf{compact}_\ell\Big) = \frac{\text{Eqn. (7)}}{\text{Eqn. (3)}}$$

V. Selecting the Attacker's Probe

In this section, we demonstrate how the adversary can use the switch model developed in Section IV-B to select optimal probes for our flow reconnaissance attack.

A. Single query

We start with the simple case where the attacker chooses a single best probe flow, i.e., one probe flow to best determine whether a target flow occurred within the last T time steps. Then, we will extend the strategy to multiple probe flows.

We use an indicator random variable $\hat{X}$ to represent whether the target flow $\hat{f}$ occurred in the last T steps, i.e., $\hat{X} = 1$ if $\hat{f}$ occurred in the last T steps and $\hat{X} = 0$ otherwise. The entropy of $\hat{X}$ is

$$\mathbb{H}\left(\hat{X}\right) = \sum_{x \in \{0,1\}} \mathbb{P}\left(\hat{X} = x\right) \log \frac{1}{\mathbb{P}\left(\hat{X} = x\right)}$$

A smaller entropy indicates a more skewed distribution of the flow's presence, and thus the attacker has more *a priori* knowledge about whether $\hat{f}$ happened or not. In particular, since we assume that the adversary knows (or can estimate) the Poisson parameter $\lambda_{\hat{f}}$ of the target flow $\hat{f}$, he can calculate the probability of flow absence as $\mathbb{P}\left(\hat{X} = 0\right) = e^{-\lambda_{\hat{f}} T\Delta}$, and therefore the entropy $\mathbb{H}\left(\hat{X}\right)$.

The attacker can probe the switch using a timing attack to reduce the uncertainty about the rules in the switch cache, as discussed in Section III. To incorporate the knowledge the attacker gains in this way, we consider the conditional entropy of $\hat{X}$ given the results Q_f of querying the cache with a specific flow f (perhaps forged as discussed in Section III). Here, Q_f is an indicator random variable showing whether the attacker's flow matched a rule already in the switch cache, which the adversary can infer from the response time to its flow; i.e., $Q_f = 1$ if f matched an already-cached rule, and $Q_f = 0$ otherwise. Then, we consider the conditional entropy

$$\mathbb{H}\big(\hat{X} \;\big|\; Q_f\big) = \sum_{\substack{x \in \{0,1\} \\ q \in \{0,1\}}} \mathbb{P}\left(\hat{X} = x \wedge Q_f = q\right) \log \frac{1}{\mathbb{P}\big(\hat{X} = x \;\big|\; Q_f = q\big)}$$

The conditional entropy shows how much uncertainty that the adversary still faces after learning whether its chosen probe

flow matched a rule in cache or not. Naturally, the adversary selects the probe flow f to maximize its information gain, i.e.,

$$\mathbb{IG}\big(\hat{X} \mid Q_f\big) = \mathbb{H}\left(\hat{X}\right) - \mathbb{H}\big(\hat{X} \mid Q_f\big)$$

To compute the information gain of each possible probe flow f, we need to compute $\mathbb{P}\left(Q_f = q\right)$ and $\mathbb{P}\big(\hat{X} = x \mid Q_f = q\big)$ for each such flow. First, we use our switch cache model to build a transition matrix $\mathbb{A}$, with each element $\mathbb{A}[\ell, \ell']$ being the probability of $\mathsf{compact}_\ell$ transitioning directly to $\mathsf{compact}_{\ell'}$, as computed in Section IV-B. The final distribution of the switch cache state can be calculated as

$$I_T = \mathbb{A}^T I_0 \tag{8}$$

where I_T is a vector representing a distribution over the cache state $\mathsf{compact}_T$ and I_0 is the distribution of the cache state initially.

Based on the final distribution I_T, the attacker can compute $\mathbb{P}\left(Q_f = 1\right)$ for a candidate probe flow f by adding the probabilities of all cache states including any rule that covers f (and $\mathbb{P}\left(Q_f = 0\right)$ by adding probabilities for those that do not). In addition, $\mathbb{P}\left(\hat{X} = 0 \wedge Q_f = q\right)$ can be calculated by repeating this calculation with a matrix $\mathbb{A}$ in which the probability of transitioning from any state by the occurrence of the target flow $\hat{f}$ is set to zero and other edges left unchanged. Finally, $\mathbb{P}\big(\hat{X} = 0 \mid Q_f = q\big) = \frac{\mathbb{P}\left(\hat{X}=0 \wedge Q_f = q\right)}{\mathbb{P}(Q_f = q)}$ and $\mathbb{P}\big(\hat{X} = 1 \mid Q_f = q\big) = 1 - \mathbb{P}\big(\hat{X} = 0 \mid Q_f = q\big)$.

B. Extension to Multiple Probes

We now discuss how to extend the above method to enable the attacker to compute multiple probe flows $f_1, \ldots, f_m$. In the model we consider here, the attacker chooses $f_1, \ldots, f_m$ non-adaptively. The attacker will choose $f_1, \ldots, f_m$ with largest information gain $\mathbb{IG}\big(\hat{X} \mid Q_{f_1}, \ldots, Q_{f_m}\big)$. The computation is similar to the one described in the previous section, with one exception: since each probe flow actively changes the switch cache state, the final distribution I_T has to be adjusted incrementally according to the previous probe flows (by introducing new rule or resetting the timeout clock for a rule).

By selecting a sequence of probe flows, the adversary actually constructs a decision tree with each layer corresponding to an attack flow. The leaf nodes of the tree are the decisions whether the flow $\hat{f}$ occurred or not according to the conditional distribution $\mathbb{P}\big(\hat{X} \mid Q_{f_1}, \ldots, Q_{f_m}\big)$. Building the classifier can help the attacker not only choose the optimal probe flows, but also select the target flow with higher information gain.

VI. Evaluation

In this section, we detail the implementation of our attack and show that:

- The use of our model improves the accuracy of these attacks by about 2% on average. However, for certain subclasses of rule sets and flow rates, this improvement can grow to 23% or more, yielding an average accuracy approaching 85%.
- Our model enables the attacker to accomplish near-optimal inference even when using sub-optimal (but still possible) probes.

A. Setup

We evaluated our attack in a network setting built by Mininet [1]. To create the network topology, we used the real-world Cisco router configurations from Stanford University's backbone network [13] as our dataset. We leveraged the Mininet API and Python to construct the Stanford backbone topology with 16 switches in total. In addition, we leveraged the Ryu controller [4] to reactively install rules on switches.

We set the number of rules as $|\mathsf{Rules}| = 12$, the switch cache size as $n = 6$, and the number of possible flow identifiers as 16.[3] Each flow is identified by a specific source IP address. Therefore, we created 16 hosts and assigned each one with one specific IP address (i.e., 10.0.1.0 to 10.0.1.15). These hosts are connected to the same port of one randomly selected switch and only allowed to send packets to a common destination host (connected to another switch) with IP address 10.0.1.16. (Imagine a client-server architecture.) Moreover, we randomly selected the Poisson parameter λ_f for each flow f (i.e., for each source host) and TTL time for each rule rule_j. Specifically, we selected λ_f uniformly from the range $[0, 1]$, and t_j for each rule_j uniformly from $\{\lceil\frac{1}{10\Delta}\rceil, \lceil\frac{2}{10\Delta}\rceil, \ldots, \lceil\frac{9}{10\Delta}\rceil, \lceil\frac{1}{\Delta}\rceil\}$. Each source host ran a background Python script to randomly select the time to send packets based on the chosen Poisson parameter and used Scapy [5] to craft and deliver the packets. Here, the hosts leveraged the ICMP protocol to deliver packets, which requires the destination host to send back a reply message. The attacker was set as another host co-located with the source hosts. It selected the optimal probe as described in Section V, injected the probe into the network, and measured the timed to observe the reply message.

To manage the rule deployment, we leveraged the Ryu SDN controller. Ryu supports wildcard rules using netmasks (i.e., rules matching multiple flows). For 16 hosts with contiguous IP addresses, there are 81 possible rules (involving up to 4-bit masks). To build the flow-rule relations (i.e., to decide if $f \in \mathsf{rule}_j$), we randomly picked 12 rules from all the 81 possible ones based on the uniform distribution. Since our rules were strictly limited to the ICMP protocol, we pre-installed a rule instructing the switches to send all the unmatched ICMP packets to the controller. Therefore, the controller could reactively install the rules upon the arrival of new flows. To permit packets used for other protocols (e.g., ARP, LLDP) to pass through the switches correctly, we proactively installed a default rule with the lowest priority and no timeout to flood all packets. We also pre-installed a rule (also without timeout) that instructs the switch to forward ICMP echo reply messages to their destinations. Note that, instead of directly setting the switch flow table size as 6, here we set it as 9 to reserve three slots for the pre-installed

[3] Obviously, 16 flow identifiers is far too small for a real network. Rather, these 16 flows identifiers can represent classes of flows for which the Poisson parameter λ_f is the rate of all flows in that class.

rules. Since we used OpenvSwitch [2] and the switch will not evict the rules without timeouts, our pre-installed rules cannot be evicted by new incoming ones. Finally, the target flow $\hat{f}$ was chosen uniformly from all flows for which the probability of absence is within a specific range (defined by the experiment parameters). We refer to the Poisson parameters, flow-rule relation, and target flow so chosen as a "network configuration."

To implement the attack, we first made the hosts randomly generate packets for a duration of 15 seconds, and then let the attacker choose its probe flow to be either simply the target flow or the flow recommended by our methodology described in Section V. To build our proposed model, we used both MATLAB R2015a and C++. The input of the attack model is the flow-rule relation (i.e., which flows are covered by each rule rule_j), the Poisson parameter λ_f of each flow f, the switch cache size n, and the TTL time t_j of each rule rule_j. Leveraging the techniques described in Section IV-B, the code first generates the transition matrix $\mathbb{A}$ of the network model. Then, the state probability distribution given an arbitrary time window can be calculated through Eqn. (8). Based on the switch state distribution, the attacker can compute the information gain of each probe flow as discussed in Section V and select the optimal one. All computations reported here were performed in a server with 2.3GHz cores and 128GB of RAM.

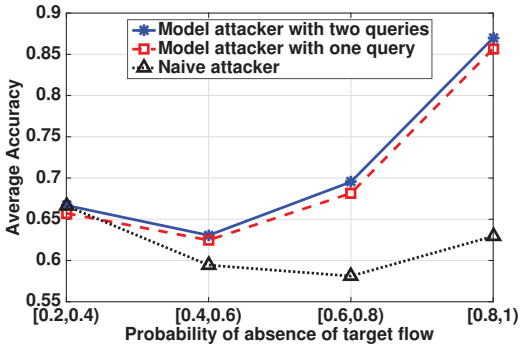
In our test platform, the latency for forwarding a packet that arrived at a switch when a covering rule was already installed was easily distinguishable from the latency when one was not. The mean and standard deviation of the attacker's observation in the former case (i.e., a covering rule was already in the switch) were 0.087ms and 0.021ms, respectively, while the values for the latter case were 4.070ms and 1.806ms. The attacker could easily differentiate these two cases by comparing the observed delay with a threshold (e.g., 1ms).

B. Attack Results

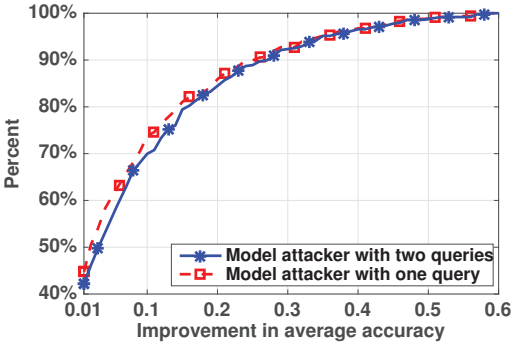
Because in some network configurations, even the selection of an optimal probe flow f provides no real information about the target flow $\hat{f}$, we limited our attention to network configurations for which our calculated $\mathbb{P}\big(\hat{X} = 0 \mid Q_f = 0\big) > 0.5$ and $\mathbb{P}\big(\hat{X} = 1 \mid Q_f = 1\big) > 0.5$. Intuitively, this means that the optimal flow can serve as a detector for the target flow. (An attacker would presumably not use our detection method on a network configuration not meeting this condition.) Under these conditions, to infer whether the target flow occurred in the last $T = \lceil \frac{15}{\Delta} \rceil$ steps, our model-based attacker made its decision by calculating the optimal flow f and returning the result of query f (i.e., Q_f). Similarly the naive attacker simply returned $Q_{\hat{f}}$.

We describe our results by considering two natural cases. In the first, we limit our attention to configurations in which the model-calculated optimal flow $f \neq \hat{f}$, i.e., configurations in which the model attacker and the naive attacker behave differently. The results for these network configurations are shown in Figure 6. Each graph represents 100 random network configurations satisfying this condition. Here, the average accuracy is defined as the ratio of the total number of true positive and true negative cases to the overall number of trials. Each test in the figure was performed 100 times, randomly generating the network packets every time. Figure 6a shows the accuracy of each attacker based on the probability of absence of target flow; as this graph shows, across the spectrum of possibilities, the model-based attacker outperforms the naive attacker by roughly 2% on average. The improvements are somewhat more impressive as the probability of the absence of the target flow $\hat{f}$ (in the last T steps) grows. Indeed, Figure 6b shows the additive improvement in average accuracy allowed by our model per network configuration, over that provided by the naive attack. As shown there, our model permits a 15% or larger improvement in average accuracy for about 20% of network configurations, and for 5% of configurations this improvement exceeds 35%.

A different view on the effectiveness of our model comes in Figure 7. Here, we no longer restrict the configurations considered to those for which the optimal probe f is different from the target flow $\hat{f}$, but instead simply restrict the model

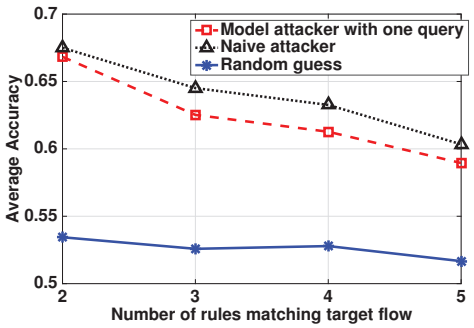


(a) Average accuracy based on the probability of absence of target flow

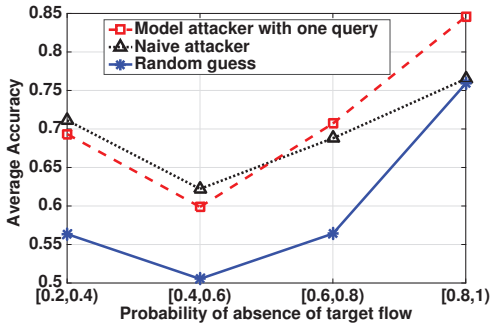


(b) Cumulative distribution function of additive improvement over naive attacker in average accuracy, per network configuration

Fig. 6: Accuracy of compact-model attacker on configurations in which model-calculated optimal probe flow f is not the target flow. (The second flow queried by the model attacker, in the two-query case, might be $\hat{f}$.) Attack is effective if it achieves better accuracy than probing $\hat{f}$ ("naive attacker").



(a) Average accuracy based on the number of rules covering target flow



(b) Average accuracy based on the probability of absence of target flow

Fig. 7: Average accuracy of compact-model attacker, when probing using flow $f \neq \hat{f}$ that provides the highest information gain, as described in Section V. Attack is effective if it achieves the same or better accuracy as probing $\hat{f}$ ("naive attacker").

attacker to not query $\hat{f}$ even if it is the optimal choice. These experiments thus reflect scenarios in which the attacker is precluded from selecting $\hat{f}$ even if $\hat{f}$ is the optimal choice, e.g., because forging this flow would raise an alert or because the attacker is at the wrong vantage point to measure the time it would take to route $\hat{f}$. In these scenarios, our goal is to do as well as querying $\hat{f}$ would have been, and as can be seen in these graphs, our model attacker does so. Moreover, it does much better than the random attacker who simply chooses whether the flow occurred based on its a priori probability of having done so, without making any probes to the network.

VII. Discussion

A. Limitations

Our work demonstrates a novel use of SDN-based timing side-channel attacks and, through the development and evaluation of a model for SDN switches, shows that these attacks can be effective. Our work has several limitations, however.

1) Reactive rule deployment: Our model is premised on a reactive model for pulling rules from the controller; i.e., the controller deploys rules only in response to a packet forwarded to it from a switch. However, SDN rule deployment can also be done proactively, not in response to a particular packet.

2) Consistent rule deployment: We have modeled an SDN switch to which a controller deploys (only) the highest-priority rule covering a flow that arrives at the switch and for which the switch does not already have a rule covering that flow. Similarly, our model reflects a switch that expires and evicts rules without attention to their overlap with other rules. However, some proposals for deploying/removing rules to/from switches do so more collectively, i.e., managing the rules in aggregate to ensure a type of consistency [16]. For example, the eviction of rules with higher priorities might cause the deletion of overlapping rules with lower priorities.

3) Scalability: Though our compact model (Section IV-B) that forms the basis for our experiments is considerably more scalable than our higher-fidelity model (Section IV-A), its ability to scale to large rule sets remains limited. That said, our tests reported here, albeit of limited size, have already revealed interesting facets of rule-set structure that can enable the attacker to conduct flow reconnaissance.

B. Countermeasures

In this section, we propose several possible defenses against this attack and list the benefits and downsides.

1) Adding delays: As suggested by Cui et al. [9], switches can delay the first few packets of each flow, even if the flow matches an existing rule in the switch, to hide that it did so. This method is easy to implement, but it increases buffering and the delay suffered by each flow.

2) Proactive rule setup: The controller can proactively install all rules on the switch during the setup phase (if there is capacity). Since the matching rules are always in the switch, the attacker cannot infer any information through probing.

3) Transform rule structure: Another defense might transform the rule structure by merging or splitting rules, increasing the uncertainty that the adversary faces after probing (our Markov model can serve as a tool to measure the information leakage of the rule structure), while maintaining the same functionality as the original rule policies. This method may steer the system toward more coarse-grained rules, raising the question of the optimal balance between coarse-grained and fine-grained control.

VIII. Conclusions

We introduced a novel flow-reconnaissance attack that can be mounted in SDN networks that use reactive rule installation. By developing a compact Markov model for an SDN switch rule cache state, we showed that an attacker can estimate the distribution over the possible switch states as flows arrive and depart. Our model tackles practical issues such as rule expirations, rule evictions, rules that overlap in the flow identifiers to which they apply, and rule priorities. We showed that our model can improve on naive flow-reconnaissance attacks by around 2% on average and up to 23% in some cases, yielding an overall flow detection accuracy approaching 85%.

Acknowledgements. The research was supported in part by NSF grant 1535917 and the Science of Security Lablet at North Carolina State University.

REFERENCES

[1] Mininet. http://mininet.org/.

[2] Open vSwitch 2.5.0 documentation. http://openvswitch.org/support/dist-docs-2.5/.

[3] Openflow Switch Specification Version 1.5.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

[4] Ryu SDN Framework. https://osrg.github.io/ryu/.

[5] Scapy. http://www.secdev.org/projects/scapy/.

[6] G. Alexander and J. R. Crandall. Off-path round trip time measurement via TCP/IP side channels. In *2015 IEEE Conference on Computer Communications*, pages 1589–1597, April 2015.

[7] D. S. Berger, P. Gland, S. Singla, and F. Ciucu. Exact analysis of TTL cache networks: The case of caching policies driven by stopping times. In *ACM International Conference on Measurement and Modeling of Computer Systems*, pages 595–596, 2014.

[8] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM*, 2007.

[9] H. Cui, G. O. Karame, F. Klaedtke, and R. Bifulco. On the fingerprinting of software-defined networks. *IEEE Transactions on Information Forensics and Security*, 11(10):2160–2173, Oct. 2016.

[10] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. SPHINX: Detecting security attacks in software-defined networks. In *ISOC Network and Distributed System Security Symposium*, Feb. 2015.

[11] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *ISOC Network and Distributed System Security Symposium*, Feb. 2015.

[12] J. Jung, A. W. Berger, and H. Balakrishnan. Modeling TTL-based internet caches. In *IEEE INFOCOM 2003*, volume 1, pages 417–426, March 2003.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[14] 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. *CoRR*, abs/1504.03095, 2015.

[15] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[16] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran. Securing the software-defined network control layer. In *ISOC Network and Distributed System Security Symposium*, Feb. 2015.

[17] S. Shin and G. Gu. Attacking software-defined networks: A first feasibility study. In *2nd ACM Workshop on Hot Topics in Software Defined Networking*, 2013.

[18] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. FRESCO: Modular composable security services for software-defined networks. In *ISOC Network and Distributed System Security Symposium*, Feb. 2013.

[19] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks. In *2013 ACM Conference on Computer Communications Security*, pages 413–424, 2013.

[20] J. Sonchack, A. J. Aviv, and E. Keller. Timing SDN control planes to infer network configurations. In *2016 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization*, pages 19–22, 2016.

[21] J. Sonchack, A. Dubey, A. J. Aviv, J. M. Smith, and E. Keller. Timing-based reconnaissance and defense in software-defined networks. In *32nd Annual Conference on Computer Security Applications*, pages 89–100, 2016.

[22] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards systematic design of enterprise networks. In *2008 ACM CoNEXT Conference*, 2008.

[23] T.-F. Yen, V. Heorhiadi, A. Oprea, M. K. Reiter, and A. Juels. An epidemiological study of malware encounters in a large enterprise. In *21st ACM Conference on Computer and Communications Security*, pages 1117–1030, Nov. 2014.