Preacher: Network Policy Checker for Adversarial Environments

Kashyap Thimmaraju[1,2] Liron Schiff[3] Stefan Schmid[1,4,5]

[1] Technische Universität Berlin, Germany [2] Humboldt Universität zu Berlin, Germany [3] Guardicore Labs, Israel
[4] Faculty of Computer Science, University of Vienna, Austria [5] Fraunhofer SIT, Germany

Abstract—**Private networks are typically assumed to be trusted as security mechanisms are usually deployed on hosts and the data plane is managed in-house. The increasing number of attacks on network devices, and recent reports on backdoors, forces us to revisit existing security assumptions and demands new approaches to detect malicious activity.**

This paper presents Preacher, a runtime network policy checker, which leverages a secure, redundant and adaptive sample distribution scheme that allows us to provably detect and localize adversarial switches or routers trying to reroute, mirror, drop, inject, or modify packets (i.e., header and/or payload) even under collusion. The analysis performed by Preacher is highly parallelizable.

We show that emerging programmable networks provide an ideal vehicle to detect suspicious network activity. Furthermore, we analytically and empirically evaluate the effectiveness of our approach in different adversarial settings, report on a proof-of-concept implementation using ONOS, and provide insights into the resource and performance overheads of Preacher.

I. Introduction

While networks are becoming programmable, faster and more efficient, they are not necessarily becoming more secure. Attackers have repeatedly demonstrated their ability to compromise switches and routers [1], networking vendors have left backdoors open [2], and national security agencies can bug network equipment and introduce hardware backdoors [3]. The attack surface on network infrastructure is further exacerbated by vulnerable implementations [1].

An unreliable data plane introduces several threats that cannot be mitigated by cryptographic communication protocols such as IPSec alone. Malicious access to the data plane [1] can result in several attacks that are damaging regardless of whether a packet's content is encrypted and/or authenticated, or not, e.g., an incorrectly mirrored packet can lead to undesired data leakage [4], or break multi-tenant isolation. Incorrectly forwarded packets may bypass firewalls or intrusion detection systems, and thereby, enter or leave unauthorized networks/hosts. To give another example, a malicious data plane can simply drop key exchange packets or route advertisements resulting in a denial of service or incorrect topology attacks respectively.

Today, we lack effective tools to verify a packet's traversal in adversarial environments. For example, while traceroute, NetSight [5] and trajectory sampling tools are useful to verify routes in *reliable networks* [6], and may still perform well in the context of faulty and heterogeneous networks [7], they are futile in adversarial environments. A compromised switch/router can report falsified information. The tools designed for adversarial environments, e.g., by Li et al. [8] have a large overhead when it comes to sampling the data plane traffic; and FOCES [9] and WhiteRabbit [10] merely uses packet counters and flow statistics respectively which cannot detect packet modifications.

Hence, we need solutions that can detect a broad spectrum of attacks (including header and payload modifications) in the presence of an adversarial data plane.

Paper Scope. This paper answers the question: How to verify whether a packet traversed the network as per the network policy? As such, our paper is orthogonal to the question of how to *prevent* misbehavior, and also complements related work [11]–[13] on topology-based defenses.

A. Basic Idea and Example

To this end, we have developed Preacher[1], a runtime-based probabilistic policy checking approach that can provably detect several types of attacks arising from a malicious data plane using programmable network technology.

A simplified view of a man-in-the-middle (mitm) attack and how Preacher detects it is shown in Figure 1. The malicious switch M in Fig. 1 conducts the mitm attack in the data plane by modifying the $A \rightarrow B$ packet to $A \rightarrow E$. As a result, the original packet is dropped at M and a new packet originates at M.

By leveraging recent networking paradigms (namely, programmability, logically centralized control, secure communication channels between the control and data plane, and the ability to sample partial or entire packets), Preacher is able to detect the mitm attack in 3 phases. In Phase 1, using a random sampling strategy, and securely distributed sampling (flow) rules installed in the data plane, Preacher obtains samples from individual switches. In Phase 2, Preacher identifies the *policy*, i.e., specific routes associated with the respective sample, and the locations of where else to expect similar samples. We will refer to the set of all policies as the *network policy*. Finally, in Phase 3 Preacher compares the policy with the received samples and those expected. In this example, Preacher detects that the $A \rightarrow B$ packet was dropped at M: since it did not receive the expected sample on the right, and the $A \rightarrow E$ packet was injected at M; and since it did not receive the expected sample on the left.

[1]Preacher stands for probabilistic and runtime-based policy checking.

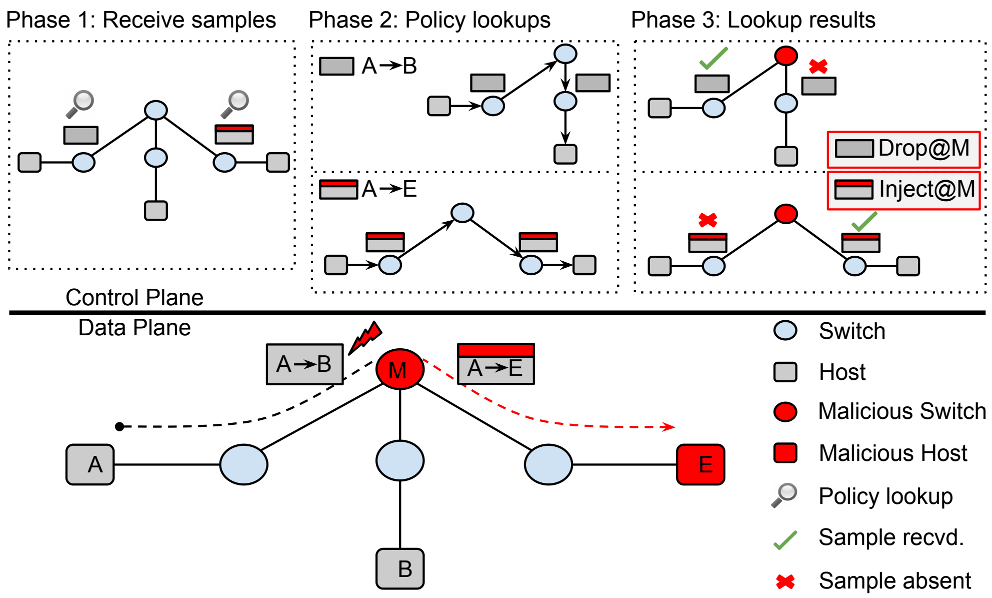


Fig. 1: Preacher detects an mitm (reroute) attack in 3 phases in the control plane. Host A sends a packet to B however, the malicious switch M, modifies the packet header (indicated by the red top) and sends it to E.

In order to ensure scalability, and since simply inspecting all packets is not practical, Preacher builds upon ideas by Lee et al. [14] and selects only part of the traffic, making the detection depend on the probability that attacked packets are inspected (in relevant locations).

B. Contributions

We present, analyze, implement, and evaluate Preacher, a monitoring scheme that detects a broad range of adversarial attacks and violations of network policies. In particular, we analytically derive the expected detection times of Preacher, considering network topology and scenario parameters, identify the core technologies that enable our scheme to be scalable and robust against failures and attacks, and empirically study tradeoffs between performance resource consumption.

To enable other researchers to reproduce our results and experiment with alternative scenarios, we make available our entire framework as well as the collected data at https://www.github.com/securedataplane/preacher.

C. Organization

Section II introduces our threat model and Section III describes Preacher in more details. We then derive the detection probabilities in Section IV followed by an extensive evaluation in Section V. In Section VI we discuss additional aspects of Preacher. After reviewing related work in Section VII, we conclude in Section VIII.

II. Threat Model

This section presents our basic threat model. We will later discuss how to deal with even stronger adversaries.

The network consists of a set of *switches* (or for the purpose of this paper equivalently: routers), connected by a set of *links*. We focus on an adversary whose target is a high-value asset, e.g., intellectual property in a company, classified government documents, etc. The switches can be compromised, e.g., the adversary may compromise the supply chain [3], [15], exploit zero day vulnerabilities [1], use social-engineering techniques such as phishing, or is an insider.

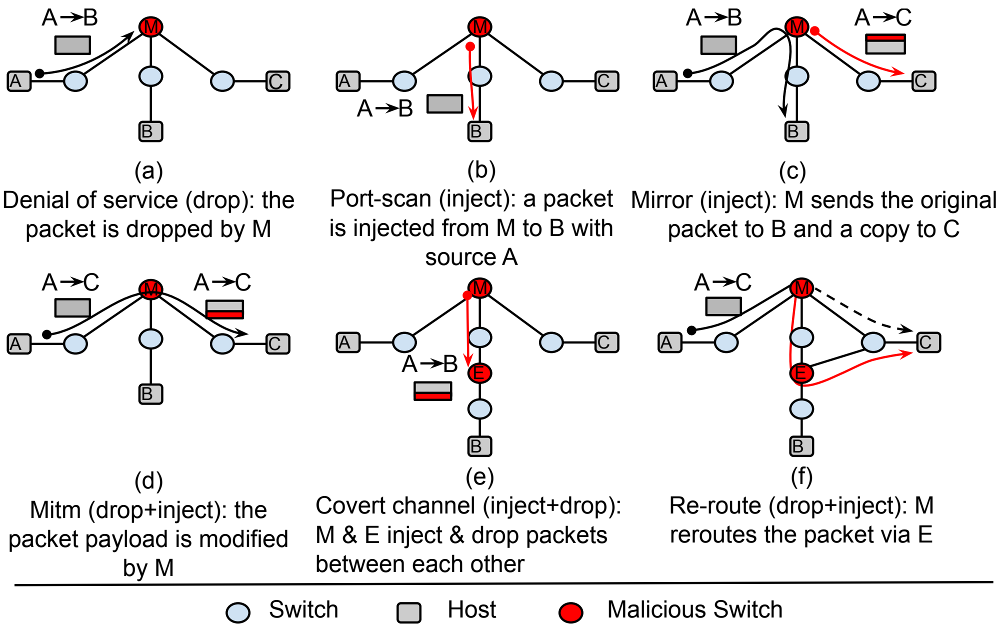
We assume a malicious switch can drop, fabricate and transmit any type of message in the data plane (e.g., duplicate packets), it can also misreport samples or statistics. If packet contents are encrypted, the attacker can exploit side channels or traffic analysis [16] to exfiltrate sensitive information, e.g., IP addresses, that can later be used to launch a targeted attack. For simplicity, in the next sections, we assume that the edge switches are trusted.

There may be more than one adversarial switch, and adversarial switches may even collude. For covert switch-to-switch communication, one switch can inject a packet and the other drops it; or, one switch does not report the to-be-sampled packets of the colluding switch. We limit the paper scope, by not considering covert timing channels and in-band control plane attacks, although we discuss how Preacher could be used to detect them in Section VI.

Furthermore, we assume that adversarial switches cannot compromise the control plane components such as the controller or Preacher's servers, however we detail in Section III-E the distributed properties of our design supporting more secure deployments that operate even in cases that some the components are compromised.

More systematically, Figure 2 illustrates a comprehensive set of attacks an adversarial switch may perform. All attacks considered in this paper can be described by combining and repeating the following two simple primitives:

1) **Drop:** An adversary prevents a packet from being sent (from one or more ports).



(a) Denial of service (drop): the packet is dropped by M

(b) Port-scan (inject): a packet is injected from M to B with source A

(c) Mirror (inject): M sends the original packet to B and a copy to C

(d) Mitm (drop+inject): the packet payload is modified by M

(e) Covert channel (inject+drop): M & E inject & drop packets between each other

(f) Re-route (drop+inject): M reroutes the packet via E

Fig. 2: A comprehensive overview of attacks, their respective primitives, and a short description of the attack.

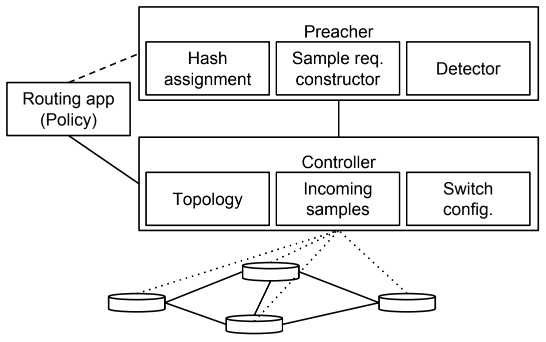


Fig. 3: Illustration of Preacher components and relation to existing network services.

2) **Injection:** An adversary fabricates and sends a new packet or resends a packet sent earlier. This also includes sending a packet from an unintended port.

III. Preacher

Preacher securely and probabilistically samples packets in the data plane to discover the actions of malicious switches (as described in Sec. II) at runtime. We first provide an overview of our approach, and then describe the technical details of Preacher that enable the detection of several attacks.

A. Preacher Overview

To sample packets, a logically centralized controller assigns a set of random hash values to every switch in the network as shown in Figure 3. If a packet passing through a switch hashes to a value assigned to that switch, it sends the packet to the controller. Each sample includes the packet, the sampling switch's id, and port id through which the packet entered the switch.

Upon receiving a sample, Preacher computes the sample's *requirement*, i.e., other related samples it expects based on the network policies, configuration and sampling assignment. For example, a requirement can be: sample the same packet along a path from its source to destination. In general, this requirement can be an optional combination of several paths, e.g., as a result of a network path load-balancer.

Given the requirement, Preacher matches it against received samples. If the requirement is not fulfilled an alert is generated by the detector, e.g, if samples are missing before or after the received sample, an injection or drop alert is reported respectively.

When a malicious switch deviates from the policy for some packet, it may be detected if that packet is sampled by other switches. Since the hash values for each switch are randomly chosen and securely distributed, a malicious switch cannot infer what other switches sample to avoid detection. The more deviations it generates, the higher the detection probability.

B. Sampling

There are 3 main aspects to sampling: *Hash function*, *Assignment* and *Sample collection*, which we elaborate in the following.

1) Hash Function: Sampling is based on a hash function used by the switch to map packets to hash values. As it is infeasible to check all packets traversing each switch in the network, we use random sampling to inspect only a predefined ratio of the available hash range defined as the sampling ratio, p_s.

- **Packet header hashing:** To apply the hash function on the packet header, OpenFlow switches can be configured to use the group-table [17] to select action buckets based on the hash of the packet header. Among these buckets, a subset of buckets are defined to send the packet to the collector. The indices of the sampling buckets are chosen according to the assigned hashes.

- **Packet payload hashing:** To apply this hash function the switch is configured to match the TCP/UDP checksum field of packets. Matching the checksum field alleviates the overhead of hashing the payload at the switch. To extend the OpenFlow match field to the checksum, the approach followed by Afek et al. [18] can be adopted. Alternatively, we can use P4 enabled switches to match the checksum fields or parts of the payload itself [19]. We emphasize that the TCP/UDP checksums are only used at the switch for sampling; deeper payload verification can be performed at the controller.

2) Assignment: The assignment process is used to randomly assign and deterministically configure switches with hash values for sampling. The assignment should form collisions: Detection depends on the probability that the same packet is (supposed to be) sampled before and after a potential attack location. This allows Preacher to accurately compute which samples are expected from every switch (but cannot be inferred from one switch to another). Preacher can offer many flexibilities in terms of hash assignment which we describe next.

- **Pairwise:** For each pair of switches, $s_i, s_j \in S$, we assign a randomly selected subset of hash values $A(s_i, s_j)$. In total, each switch s_i is assigned with the union of all subsets selected for all its pairs, i.e., $A(s_i) = \cup_{j \neq i} A(s_i, s_j)$. In the pairwise approach we ensure collisions (with probability 1).
- **Independent:** Each switch is assigned with a randomly selected subset of hash values, independently from other switches. Although the expected collision ratio can be controlled, in some cases this scheme may result in ineffective assignments, e.g., locally unmonitored switches.
- **Dynamic update:** It is useful to change the hash assignment on the switches as at any moment in time, Preacher samples only a fraction (p_s) of the traffic and the hash function may not be uniform for the given traffic distribution. In this way, over time, it becomes difficult for the adversary to avoid being detected as no information about a static sampling pattern is leaked. Such a strategy can be applied regardless of using the pairwise or independent schemes. Since packet arrivals and switch configurations are inherently asynchronous, verifying samples during assignment updates are non-trivial and may introduce false-positives. Determining the exact time at which a new configuration takes effect is imprecise and variable across switches [20]. To avoid false positives, Preacher conservatively suppresses alerts related to the updated switch and hash values. In particular, hash values are changed at *random times* (update rate) and involve only one/two (pairs of) switches (update size) at a time: thus reducing the attacker's ability to abuse uncertainty.

3) Sample Collection: Samples can be collected in parallel and dispatched to the detector to support high throughput and load-balancing. The samples are securely delivered to the controller as *Packet-in*s, avoiding sample integrity issues [14]. In the case of distributed controllers, the *Packet-in*s can be sent to the same controller which configured the sampling rules, or to a dedicated controller depending on the operator's requirements. In Section III-E, we discuss how a distributed setting can be used to increase the security against attacks on Preacher itself.

C. Detection

The detection in Preacher comprises of two parts: sample requirement and sample processing.

1) Sample Requirement: At the heart of our system lies the parallel construction and fulfillment of a sample *requirement*. A sample requirement for a (sampled) packet is the set of all samples expected at the controller as the sampled packet traverses the network. The requirement also includes a (partial) order relation of the samples indicating the order in which the samples should be generated, which is important for the analysis of attacks. Note that this order may differ from the actual order the samples arrive at the collector. As we will see, an unfulfilled requirement raises an alert, e.g., a drop or inject attack.

Sample requirements can be computed at the controller as it has the global network policies, configuration and topology to compute the set of all possible samples (recall Sec. III-A, a sample consists of the packet, the switch-id and in-port) that can be generated as the packet traverses the network. Switches not expected to generate samples are filtered out using the hash assignments.

In simple network policies, where forwarding actions move packets from an ingress to an egress port, the set of expected samples is along a path from the packet's source to destination. However Preacher can also support more advanced policies:

1) *Access Control Lists (drop rules):* Samples are not expected after the (legal) drop point.
2) *Multicast/Broadcast (packet duplication rules):* Samples are expected at multiple branches after each duplication point.
3) *Network load-balancing (random path):* Samples are expected along exactly one path out of a few possible paths.

Note that the above policies do not modify the packet. Section III-D discusses support for modifying policies.

2) Sample Processing: To scale Preacher, samples can be processed in parallel by load-balancing them across several *detection* threads, e.g., based on their hash values that were assigned to the switches. The steps to detect an attack using the samples and sample requirements are shown in Algorithm 1.

Each thread adds the sample smp, and the corresponding $timestamp$, from the incoming sample queue Q, to its history list $History$, where the samples wait till a $timeout$ expires. The $timeout$ is used to ensure that enough time has passed for the other expected samples to arrive. Samples are then removed from the $History$ to compute their respective requirement. Using the $Policy$, we get the expected samples $traversal\_set$, and their (partial) order ord. The $traversal\_set$ is then filtered using the $Assignment$. Next,

Alg. 1 Detection

```
Require: assignments $Assignment$, switches $S$, samples queue $Q$, requirement policy $Policy$
 1: $History \leftarrow ()$                                        ▷ empty list
 2: $t_0 = time()$                                                  ▷ current time
 3: while $true$ do
 4:     $timestamp, smp \leftarrow Q.get()$                         ▷ blocking get
 5:     $History.add(timestamp, smp)$
 6:     if $timestamp - t_0 < timeout$ then
 7:         continue
 8:     while $History.min() < timestamp - timeout$ do
 9:         $timestamp', smp` \leftarrow History.get\_min()$
10:         $traversal\_set, ord \leftarrow Policy.get\_possible\_samples(smp')$
11:         $smp\_set \leftarrow Assignment.remove\_unassigned(traversal\_set)$
12:         if $\exists smp1, smp2 \in smp\_set$: $smp1 <_{ord} smp2 \wedge smp1 \in History \wedge smp2 \notin History$ then
13:             Report Drop of $smp'.pkt$ between $(smp1.s, smp2.s)$
14:         if $\exists smp1, smp2 \in smp\_set$: $smp1 <_{ord} smp2 \wedge smp1 \notin History \wedge smp2 \in History$ then
15:             Report Injection of $smp'.pkt$ between $(smp1.s, smp2.s)$
16:         for $smp \in smp\_set$ do
17:             $History.remove(smp)$
```

the remaining samples, smp_set, are checked against the $History$ and then removed from the $History$ as follows:

- *Drop attack:* A drop is reported if the requirement of receiving a succeeding sample ($smp2$) is unfulfilled in the $History$.
- *Inject attack:* An inject is reported if the requirement of receiving a preceding sample ($smp1$) is unfulfilled in the $History$.

In the case the policy dictates a path, it suffices to check the required samples one by one according to the order, comparing consecutive switches.

While the mechanism to identify injection and drop events are similar, the severity of, and reaction to these two events may differ. In particular, while injections may occur rather rarely *by accident*', benign packet drops occur often. Accordingly, for drops arising individually and without statistical patterns, no alarm should be raised. To deal with the ephemeral hash value assignments and avoid false positives, we introduce a grace period around dynamic updates.

3) Locating the attack point: While our algorithm does not directly locate the switch that carried out the attack, it is possible to infer the attacker's location over time by intersecting the reported attack intervals. An attack interval is the sets of switches (from $traversal\_set$) ordered between (and including) the pair of switches reported by the detection algorithm. Note that the longer the attack, the higher the probability it will be detected by its immediate neighbors (see Section IV-B1). Moreover, this localization approach does not make additional switch configuration changes in response to the detected attack and as such is harder to be avoided by the attacker.

D. Handling Packet-Modifying Policies

In some cases the network policy may require the switch to modify the packet's header (e.g., decrease TTL, add MPLS label, rewrite IP and port). Some of these policies can be modeled as a function that receives as input, a packet, a switch id and an ingress port id, and returns a modified packet and an egress port id. Applying this function multiple times from the moment a packet enters the network till it leaves, reveals the locations and values of the packet in the network. However for Preacher to detect attacks in a network it should be able to also compute past locations and values of a packet given a sample of it from an arbitrary location in the network. This requires Preacher to apply the function in reverse (e.g., increment TTL, remove MPLS label, reconstruct original IP and port before NAT).

While not all network policies can be modeled and reversed in this way, it may be possible (e.g., in a software-defined network), to associate a packet with a specific service and routing directive configured for it (e.g., MPLS path) and to use that to infer the packet path and modified values. Moreover, sampling should be aware of (legal) policy updates, e.g., in the pair-wise assignment if packet hashes are affected by the modifications, the packet may not be sampled as expected. Therefore, we suggest payload based hashing which is not affected by header modifications.

E. Handling Control Plane and Preacher Targeted Attacks

Attacks targeting Preacher components can be used to undermine its security guarantees. For example, gaining access to the hash assignment allocator allows the attacker to drop/inject packets that are not monitored. In addition to standard security measures that can protect (any) computer system, we explain how Preacher components, hash assignment and samples verification, can be distributed and their traffic encrypted, to provide security and resiliency guarantees.

As discussed in Section III-B3, samples are delivered securely to controller(s). This can be achieved by encryption and authentication, both are recommended for network administration protocols and are supported by programmable networks. A greater threat is introduces for in-band control planes which we address in Section VI-B.

Regarding the configuration of hash assignments and the verification of samples, redundancy is key to overcome failed or compromised component(s). As discussed in Section III-C, different detector instances can be used to verify samples. In addition, hash assignments can be distributed to multiple independent hash assigners, where each assigner configures a partial assignment. Redundancy exists among the assigners and the verifiers as each switch pair should be assigned a hash value by more than one hash assigner and each sampled packet should be reported to more than one detector.

Furthermore, each assigner should be allowed to read and update only switch rules related to its own assignment (otherwise one assigner can gain access to all assignments through the switches). Such a constraint can be enforced by the (distributed) controller.

IV. Analysis

To prove the effectiveness of our detection algorithm (Alg. 1), we now derive the probabilities and expected times for detection of attacks, from single packet to complete flow attack with one and more attackers.

We focus on the payload hashing function and hash assignment where all switches have the same sampling ratio, p.

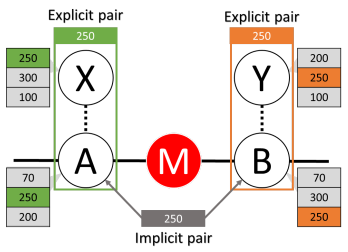


Fig. 4: Illustration of explicit and implicit pairs in a network with a pairwise assignment. The attacker's (M) position is (1,1). Explicit assignments for pairs $\{X, A\}$ and $\{Y, B\}$, indicated by the green and orange rectangles respectively, have the same value 250, making them an implicit assignment for pair $\{A, B\}$. For simplicity, not all explicit and implicit pairs are indicated.

With pairwise assignment, for each pair of switches we assign $p/(n-1)$ of the hash space $|H|$ (independently at random from the other pairs), where n is the total number of switches in the network. However, with independent assignment, for each switch we directly assign p of the hash space $|H|$ (independently at random from the other switches).

We use the notation (k_1, k_2) to indicate the number of switches before and after the attacker resp. along a path.

A. Single Packet Attack

We assume that the attacked packet traverses a single path and the attacker's location is (k_1, k_2) along that traversal. A detection occurs if the packet hash is assigned to at least one switch before, and at least one switch after the attack location. By definition, there are exactly $k_1 \cdot k_2$ assignment pairs surrounding the attack location (k_1, k_2): denoted as *explicit pairs*.

However, the detection probability is slightly higher than only considering explicit pairs. Due to a *birthday paradox*, it is probable that two assigned (explicit) pairs, e.g., $\{A, X\}$ and $\{B, Y\}$, accidentally assigned the same hash value thereby forming *implicit pairs*, e.g., $\{A, B\}$, that also surrounds the attack location (see Fig. 4).

Considering attack location (k_1, k_2), implicit pairs can be formed from explicit pairs involving the k_1 switches intersecting explicit pairs involving the k_2 switches (regardless of any pair involving both of them). The number of pairs involving k_1 but not k_2, K_1^{-2}, is $\binom{k_1}{2} + k_1 \cdot (n - k_1 - k_2 - 1)$. Similarly, The number of pairs involving k_2 but not k_1, K_2^{-1}, is $\binom{k_2}{2} + k_2 \cdot (n - k_1 - k_2 - 1)$.

The detection probability P_{pa}, equals the probability that the packet is sampled by at least one of the pairs (explicit or implicit) surrounding the attacker.

$$P_{pa} = 1 - (1 - q)^{k_1 \cdot k_2} \left(1 - \left(1 - (1 - q)^{K_1^{-2}}\right) \left(1 - (1 - q)^{K_2^{-1}}\right)\right) \quad (1)$$

where we define $q = \frac{p}{n-1}$ which is the probability that a hash value belongs to a pair assignment.

Alternatively, when the independent hash assignment is used, the detection probability, P_{ia}, simply equals the probability that a hash value belongs to some switch before and some switch after the attack point.

$$P_{ia} = \left(1 - (1 - p)^{k_1}\right) \cdot \left(1 - (1 - p)^{k_2}\right). \quad (2)$$

Note that the detection probability in this scenario does not depend on the network size (n). However the maximum sampling probability (p) that can be handled is usually inversely proportional to network size. We will discuss the tradeoff between required resources vs detection time in Section V-F. In case the attacker also attacks packets that she samples, the attacker can be included in k_1, i.e., k_1 increases by 1.

Theorem IV.1. *If the sampling ratio is $0 < p < \frac{1}{3n}$ then the detection probability using a pair-wise assignment is higher than using independent assignments, i.e., $P_{pa} > P_{ia}$.*

Proof. Using Taylor polynomial approximation we get that for positive integer m and $0 \le x \le 1$:

$$1 - xm - E(x) \le (1 - x)^m \le 1 - xm + E(x) \quad (3)$$

where $E(x) = \frac{1}{2}m(m-1)x^2$. Therefore the detection probability with independent assignment can be upper bounded as follows:

$$\begin{aligned} P_{ia} &\le \left(pk_1 + \frac{1}{2}k_1(k_1 - 1)p^2\right) \cdot \left(pk_2 + \frac{1}{2}k_2(k_2 - 1)p^2\right) \\ &= p^2 k_1 k_2 \left(1 + \frac{1}{2}(k_1 - 1)p\right) \cdot \left(1 + \frac{1}{2}(k_2 - 1)p\right). \end{aligned}$$

Next, by the fact that $k_1, k_2 < n$ and $p < \frac{1}{3n}$ we get that

$$\begin{aligned} P_{ia} &\le p^2 k_1 k_2 (1 + 1/6)^2 & (4) \\ &< \frac{1}{3n} p \cdot k_1 k_2 (1 + 1/6)^2 = \frac{49}{108} \frac{pk_1k_2}{n}. & (5) \end{aligned}$$

Regarding the pairs assignment, ignoring the implicit pairs, we get that $P_{pa} > 1 - (1 - q)^{k_1 k_2}$. Then using again Inequality (3) we obtain:

$$\begin{aligned} P_{pa} &> qk_1k_2 - \frac{1}{2}k_1k_2(k_1k_2 - 1)q^2 \\ &= qk_1k_2\left(1 - \frac{1}{2}(k_1k_2 - 1)q\right). \end{aligned}$$

Next, by the fact that $k_1, k_2 < n$ and $p < \frac{1}{3n}$ and $q = \frac{p}{n-1}$ we get that

$$P_{pa} > \frac{pk_1k_2}{n}(1 - 1/6) = \frac{90}{108} \frac{pk_1k_2}{n}. \quad (6)$$

□

B. Packet Flow Attack

When the attacker attacks (drops or injects) multiple packets of the same flow they also share the same path and attack location expressed by (k_1, k_2). Assuming the packets have uniformly distributed hash values, the detection of each packet in the flow is (ideally) an independent trial, each with a success probability P which is either P_{pa} or P_{ia} depending on hash assignment. Therefore, the number of attacked packets till detection follows a geometric distribution with expected value $1/P$ and the expected detection time can be derived by the product of the average flow rate and $1/P$.

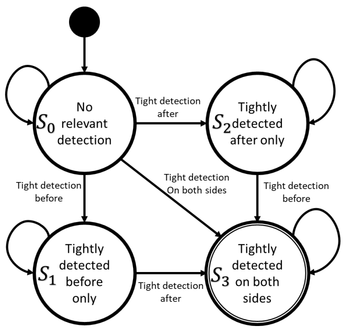


Fig. 5: Illustration of the attack localization (Markovian) states.

1) Localization: As explained in Section III-C3, we can improve the localization of the attack over time by intersecting detected attack intervals (the switches between the pair of sampling switches). Therefore, to find the exact location of attack we need the attacked packets to be sampled immediately before and immediately after the attack location. As these two conditions do not have to be fulfilled for the same packet, we can define a Markov process with 4 states (see Fig. 5): in the initial state S_0 no attacked packet has been sampled adjacent to the attack point. In states S_1 and S_2 attacked packets has been sampled only immediate before or only immediate after the attack location respectively. In final state S_3, attacked packets have been sampled immediate before and after the attack location.

Each attacked packet may change state from S_i to S_j with probability denoted by $P_{i,j}$. Clearly moving backward has probability 0, i.e., $P_{1,0} = P_{2,0} = P_{3,0} = P_{3,1} = P_{3,2} = 0$ and also moving between states S_1 and S_2 is not possible, i.e., $P_{1,2} = P_{2,1} = 0$. We denote the average time we may stay at state S_i by T_i, which is a geometric distribution with success probability $1 - P_{i,i}$, i.e., $T_i = \frac{1}{1-P_{i,i}} - 1$. In addition, we use $P'_{i,j}$ to denote the conditional probability to move from state S_i to S_j given that we leave state S_i, i.e., $P'_{i,j} = \frac{P_{i,j}}{1-P_{i,i}}$. Given the notation above, the average time to localize an attack can be expressed by $T_0 + P'_{0,1} \cdot (1+T_1) + P'_{0,2} \cdot (1+T_2) + P'_{0,3}$.

To conclude we only need to calculate $P_{0,1}, P_{0,2}, P_{0,3}, P_{1,3}$ and $P_{2,3}$, where easy to see that $P_{1,3} = P_{0,2} + P_{0,3}$ and similarly $P_{2,3} = P_{0,1} + P_{0,3}$. $P_{0,1}$ is equal to the detection probability of a single packet attack, as described in section IV-A, where we decrease k_2 by one and set k_1 to one. Similarly $P_{0,2}$ can be obtained by decreasing k_1 by one and setting k_2 to one. Finally, $P_{0,3}$ can be obtained by setting both k_1 and k_2 to one.

We valuated the localization times for different attack types (from Fig. 2) along a path of length 5 and different network sizes. We used a Markovian process based calculation and also developed a simple discrete time simulation of the attacks and Preacher. We considered an attack rate of 100pps, with uniform packet hash distribution. Moreover we considered Preacher with pair-wise assignment, with configurations providing the same sampling probability (p_s) of 0.009 for every network size. As we can see in Fig. 6, even in large networks most attacks are localized within a minute or less. The results also demonstrate the tightness of the analysis.

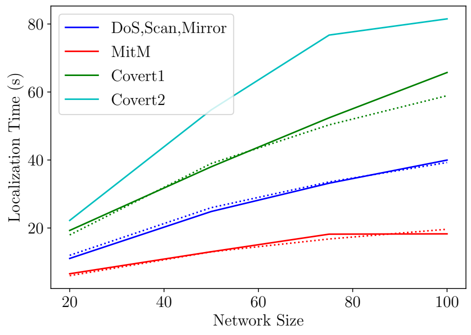


Fig. 6: Average localization times. Solid and dashed lines represent simulation and analytic results respectively. Covert1 and Covert2 represent the times till localization of one and both of the colluding switches respectively.

2) Constant Hash Flow Attack: Especially in inject attacks, the hash values of attacked packets may not distribute uniformly and independently. In the worst case, the attacker will make sure all attacked packets hash to the same value and the attack will either be detected from the first packet of the flow, or never. The initial (and static) assignment of the switches dictates the detection.

By changing the assignments across the network with new random values, at random times (following a memoryless Poisson distribution), the attack may be detected with a higher probability over time. The detection will occur when the process of updates results in an assignment surrounding the attacker which includes the hash value used in the attacked flow.

Let T_u be the average time it takes to complete a full policy update, for example in the case of pair-wise assignment, assume that updated pairs are chosen at rate λ_u (pair assignments per second) following a random permutation of all $\binom{n}{2}$ pairs, then $T_u = \binom{n}{2}/\lambda_u$. Next, let T_a be the average inter arrival time for attacked packets, i.e., $T_a = 1/\lambda_a$, where λ_a is the average attacked flow packet rate. We can upper bound the detection time by maximum between T_u and T_a divided by the single packet detection probability, P.

C. Collusion

Our analysis generalizes to collusion. If g_1 and g_2 are benign switches before and after the injection location respectively, and if b are colluding adversarial switches, we can use Equations 1 and 2 by adapting the model.

We redefine k_1 as the number of switches that are inconsistent with the attack, i.e., will not report injected packets even if assigned to do so, or will report dropped packets. Similarly we redefine k_2 as the number of switches that are consistent with the attack, i.e., will report injected packets or will not report dropped packets if assigned its hash. Benign switches before the attack location are counted in k_1 and those after the attack location are counted in k_2. Adversarial switches, including the attacker, are counted based on their attack strategy.

If adversarial switches share information about the attack and b_1 of them are inconsistent regarding the attack, e.g., they recognize packets injected before them along the path and do not report them, then they are counted in k_1 (regardless of their location along the path), i.e., $k_1 = g_1 + b_1$. Similarly, if b_2 adversarial switches are consistent with the attack, e.g., report injected packets (even before they are injected), then they are counted in k_2, i.e., $k_2 = g_2 + b_2$.

However, if colluding switches share their hash allocations and the adversary avoids attacking with hashes assigned to any of them, then these switches are not considered in k_1 nor in k_2, i.e., $k_1 = g_1$ and $k_2 = g_2$. Moreover to analyze such cases more accurately, we need to ignore all mutual (pairwise) hash allocations of other switches with the colluding switches, by reducing n by b in K_1^{-2} and K_2^{-1}, as such allocations do not include the attacked packet hash. Note that while this strategy results in lower detection probability compared to previous strategies, not all packets are applicable for such an attack.

V. Experimental Evaluation

We implemented a prototype of Preacher and conducted extensive experiments using realistic traffic workloads. In particular, we evaluated the performance of our detection algorithm, investigating both the detection time as well as the detection throughput of Preacher. Furthermore, to gain insight on possible performance overheads introduced by Preacher, we investigated both resource overheads at the collector, and forwarding overhead in the data plane.

A. Prototype

We implemented a prototype using Software-Defined Networking (SDN). Indeed, SDN in particular and programmable networks in general provide an ideal framework to implement Preacher for the following reasons:

- **Programmable, and logically centralized control:** The SDN controller provides an ideal platform to implement the instructions for sampling, as well as to receive samples.
- **Network-wide view:** The SDN controller has a global view of the network and is configured with the network policies, e.g., routes, ACLs, etc. Therefore, it can determine the intended route and transformation for every packet that traverses the network.
- **Secure communication channels:** Reliability, encryption, integrity and authentication between SDN switches and controllers are readily supported via TCP and TLS. This prevents malicious switches from eavesdropping on other switches' assignments or samples.
- **Support for sampling:** An SDN switch readily supports the necessary functionality for sampling in the form of *flow-rules* and *group-tables*. Sampling can be performed on a per-packet or per-flow granularity as packets can be matched against several header fields. For more granular and customized sampling P4 [21] can be used. Samples can be delivered to the controller as *Packet-in* messages, e.g., in P4 and OpenFlow. As we will see, by collecting entire packets and not just headers in the *Packet-in* more sophisticated attacks, e.g. payload modifications, can be detected.

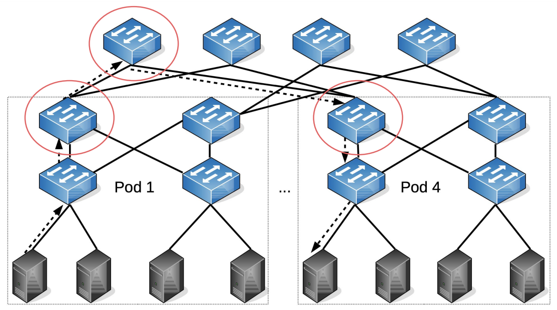


Fig. 7: Topology used for detection time, trajectory throughput and sampling overhead experiments of Preacher. Traffic flows between hosts flow from Pod 1 to Pod 4, indicated by the dotted line. The attacker locations are illustrated via the red circles.

Concretely, in our prototype implementation, we use ONOS-1.4 with OpenFlow 1.3 as our controller, and implemented the *Sampling* and *Detection* logic (recall Sec. III) as a *multi-threaded* application (see Fig. 3). We leverage the various services ONOS offers in our prototype. We use the Flow objective, Flow rule, Device services to install sampling rules in the switches, we use the Packet-in service to receive samples. For the network policy, we use a deterministic version of the forwarding app's routing algorithm. For the switches, we use Open vSwitch (OvS). For simplicity, in our test traffic we ported the checksum field into the VLAN field and matched that instead. In real scenarios we suggest the use of switches with the experimenter field support [18] or P4 support and program customized packet parsers [22].

B. Experimental Setup

While Preacher can be deployed in any network (data center, wide-area, etc.), we evaluate a Clos topology with $k = 4$, using the Ripcord platform [23] as depicted in Figure 7. For realistic network traffic, we replay LLBNL traffic [24] traces adapted to our topology. Traffic flow is uni-directional, from one host in Pod 0, to another host in Pod 3 as per the default Ripcord topology. From the concepts defined in Sec. III, our default parameters for Preacher are the following: **Detector threads** t**:** 1; **Hash Function** h**:** payload dependent; **Sampling Ratio** p_s**:** 0.4%; **Assignment:** Pairwise static and dynamic; **Dynamic Update Rate:** 2s and **Dynamic Update Size:** 2. The sampling ratio is chosen such that every switch in the network forms at least one hash collision with every other switch. In the following, we will explicitly state any changes to the default parameters and traffic.

C. Detection Time

To validate and complement our formal analysis of the mean detection time, we conducted several experiments which we describe in the following. Since the sampling ratio impacts the detection time, we measure the detection time for **Sampling Ratio** p_s**:** 0.9% and 1.3%.

1) Single Attacker: We evaluate the effectiveness of detecting the flow drop and flow injection attacks when the attacker is the aggregate switch, and when the attacker is the core switch along a single path in our topology. In the flow drop attack, the switch drops all packets from a flow along a path except those that it is meant to sample. In the flow inject attack, the switch injects a new packet in a flow along the path of an existing one with the exception of packets that it samples. Both attacks are easily emulated via OpenFlow flow rules on OvS. We count the number of packets that are sent in a flow till an alarm is raised by Preacher and then stop. We perform 100 such trials for **each attack** and **each attacker**.

Results: Fig. 8a shows the data for the pairwise static assignment from the inject attack experiments. We observed very close values for the drop attack and for the dynamic assignment as well, hence we do not show them here. The figure confirms our theoretical analysis of the attack: The theoretical means (from Sec. IV) are close to the experimental values. We observed variance in the detection, which could be due to the non-uniform distribution of the TCP checksum field in the traffic used [25]. For a fixed network topology, we observe that increasing the sampling ratio p improves the detection, roughly linearly: by doubling p_s we detect the attack in approximately half the expected number of packets. The position of the attacker also influences the detection, i.e., it takes fewer packets to detect the malicious core switch. This is because there are more pairs surrounding the core switch than the aggregate switch. Finally, the time to detection also depends on the rate of packets being attacked. For example, if the packet rate for a flow under attack by the aggregate switch is 1000 pps, then the attack will be detected in approximately 1.5 s.

2) Colluding Attackers: We evaluate the effectiveness of detecting the flow injection attack (analogous to mirroring) when two aggregate switches collude, resp. when two aggregate and one core switch collude: the switches collude to not report samples for all packets injected. We emulate that by inserting a high priority flow rule that bypasses the sampling for injected packets. In this attack, the benign traffic flows from Pod 0 to Pod 3, and the injected traffic from Pod 0 to Pod 1. The remainder of the methodology is as the single attacker (Sec. V-C1).

Results: Fig. 8b shows the data from the collusion experiments for the static assignment as we observed similar values for the dynamic assignment as well. Firstly, we observe that with fewer benign switches, it takes more packets to detect an attack. Second, we observe that detecting colluding attackers with real traffic is very close to our theoretical analysis. Finally, as the sampling ratio increases, the experimental values come closer to the expected values.

D. Detection Throughput

Next, we study the number of samples per second which can be analyzed in parallel, i.e., the detection throughput. Recall that Preacher is multi-threaded. The evaluation was carried out on a 64 bit Intel Core i7-3517U CPU @ 1.90 GHz with 4GB of RAM. Here we use the following **Detector threads** t**:** 2, 4, 6 and 8, in addition to the default parameters.

To measure the detection throughput we record the total time taken for a single sample to be dispatched to its respective detector and for the detector to complete the detection. Each detector thread makes 1k detections, from which we average the throughput for t detector threads. We repeat the measurements for different CPU core counts (1, 2 and 4/hyper-Threading).

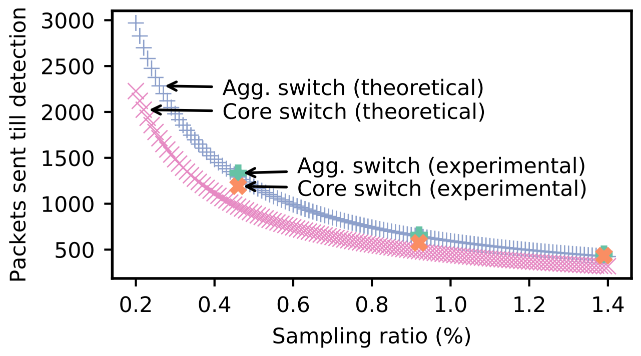
Results: The data from the experiment is shown in Fig. 6. Although we observe an increase in the throughput with more threads, it is not linear. The two main reasons for this are: (i) Reads and writes are synchronized for the History list and; (ii) ONOS and OS tasks are scheduled in addition to network interrupts. Nevertheless, the results lend credence to the use of multiple detection threads for high detection rates, and high availability. Furthermore, multiple threads on multiple cores on multiple controllers can substantially increase the detection throughput.

E. Overhead of Sampling

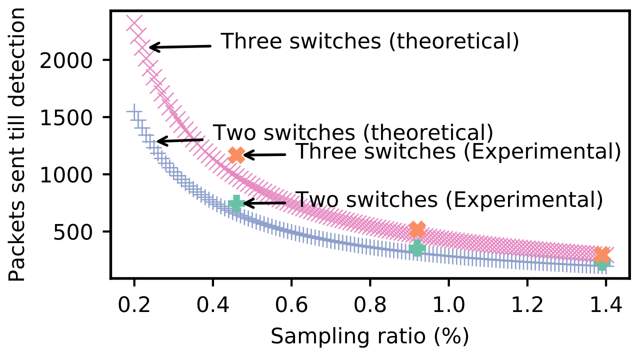
1) Overhead at the Controller: We first measure the resource consumption without Preacher (baseline), then a static assignment of Preacher, and finally the dynamic assignment of Preacher. We use the default Preacher parameters as mentioned in Sec. V-B and the same system from the detection throughput experiment. We replay bi-directional traffic at 100 pps between 6 pairs of hosts, i.e., each pod receives 3 flows, for 10 minutes. We pin the controller to one core and, use top to measure the CPU and memory every second.

Results: Naturally, Preacher does increase CPU and memory consumption on average compared to the baseline. Dispatching samples to respective detector threads, and verification are primary reasons for CPU increase. Memory increase ($\sim$1.5 MB) is due to data structures used to store and process samples, and hash assignments for switches. 50% of the CPU usage for the static and dynamic cases is within 12-35% usage, and 50% of the memory usage is within 14-15.5 MB: both are within acceptable boundaries for scenarios where Preacher executes on a controller that manages the network. Furthermore, the dynamic assignment, even with an average update rate of 2s, does not increase the mean CPU and memory usage by much. This implies that the benefits of the dynamic assignment can be reaped with minor overheads in terms of CPU and memory. Moreover, as a stand-alone system these overheads are acceptable. Hence, we view the overhead at the controller to be a small trade-off for the benefits Preacher offers: highly parallel, distributed and high-performance.

2) Overhead at the Switch: We use a dedicated server (2.5 GHz Dual Core AMD Opteron, 16 GB RAM and 4 Intel Gigabit NICs) for OvS (2.3.2), connected to a traffic generator, and a traffic sink. Passive network taps across those links allow us to collect the packets at a dedicated server (Monitor). The Monitor captures packets using dedicated queues and hardware timestamps for accurate and precise measurements.



(a)



(b)

Fig. 8: (a) Average no. of packets to detect the flow inject attack at different positions using the pairwise assignment in the static setting. (b) Average no. of packets to detect the flow inject attack when two and three switches collude.

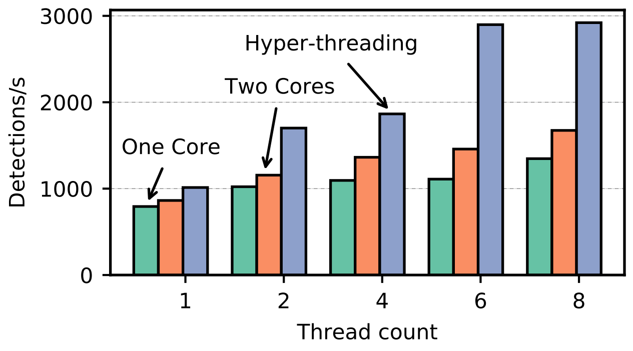


Fig. 9: Comparison of Preacher detection throughput with varying thread counts and CPU cores.

Results: We observed no impact to the forwarding throughput for different packet sizes and transmission rates. Similarly, sampling introduces little to no overhead on the overall latency of forwarding as a majority of the packets traverse OvS's fast path, which is in the kernel. Hence, we do not show these figures here.

F. Resources vs. Detection Time Tradeoffs

By applying the expected detection probability formula from Section IV to different Clos network sizes, we show the connection between the compute resources used by Preacher, in terms of inspected sampling rate, and its performance, in terms of detection time. For this analysis, we consider 3 scenarios of malicious switch activity: (a) core or (b) agg switch dropping (or modifying) all packets between two datacenter hosts and (c) agg switch dropping (or modifying) all traffic of one datacenter host with external hosts (e.g., the Internet).

We assume throughput of almost 10Gb/s (duplex) leaving each rack, of which 66% remains under the same aggregation switch, 14% flow between aggregation switches (through core switch) and 20% leaves the datacenter [26]. Moreover we assume 20 hosts per rack and consider average and uniform inter-host traffic for the attacked packet rates.

Considering system throughput of 1,000 samples per core (see our methodology in Section V-D), we can see in Figure 10 that for small networks and attack scenarios (a) and (c), Preacher running with a few cores can achieve below 10 minutes detection time while for bigger networks, tens of cores are required. Moreover for attack scenario (b) similar resources are required just to achieve below one hour detection time. This can be explained by the very short attack path in scenario (b), of length 3, compared to length 5 in (a), and low attacked traffic rate, one pair of hosts (total aggregation rate divided by number of host pairs), compared to higher rate in scenario (c) of one host to any external host (external traffic divided by number of hosts).

When comparing the performance of pairwise assignment to independent assignment for scenario (a), we can see in Figure 11 that for small networks and pairwise assignment, Preacher running with a few cores can achieve below 10 minutes detection time while for bigger networks, tens of cores are required to achieve below one hour detection time. Moreover for independent hash assignment hundreds of cores are required to achieve similar detection times even for small networks.

VI. Discussion

We now discuss how Preacher can be improved further as well as how to handle special security cases.

A. Extending Preacher

Increasing Sampling Points. Preacher comes with the fundamental property that the closer the trusted sampling points (e.g., trusted switches) are found near the flow endpoints (e.g., hosts), the tighter the security guarantees Preacher can provide. Clearly if a packet is not sampled correctly (according to the assignment) before (and after) an attack point, the attack will not be detected.

For example, in cloud management systems such as OpenStack, Open vSwitch (OvS) is used as a virtual switch and

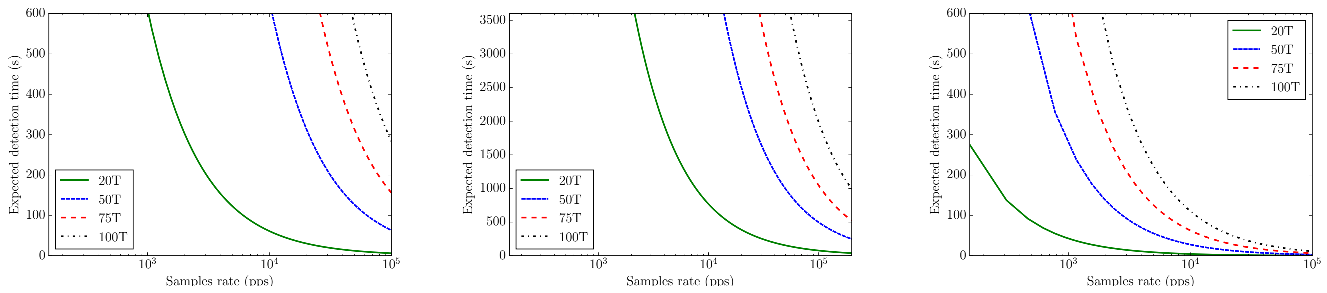


Fig. 10: Detection resources, in terms of inspected sampling rate, compared to detection time, considering different Clos network sizes (number of ToR switches in the legend) and varying the sampling probability (not shown on the graph). Three attacks are analyzed: (a) core or (b) agg switches drop (or modify) all packets between two datacenter hosts and (c) agg switch drops (or modifies) all traffic of one datacenter host with external hosts (e.g., on the Internet).

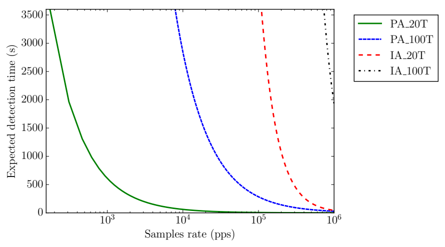


Fig. 11: Detection resources, in terms of inspected sampling rate, compared to detection time. The attack is analyzed in which a core switch drops (or modified) all packets between two datacenter hosts.

is already OpenFlow compatible. Assuming that the server's host OS/hypervisor can be trusted, OvS (or the networking stack on the host OS) can also participate in our scheme to detect misbehaving ToR switches. Alternatively, so-called *smart NICs* [27] (since 2017)–with multiple physical ports running a full fledged OpenFlow switch (OvS) on a multi-core ARM processor– can sample packets and send them to Preacher if the host OS cannot be trusted.

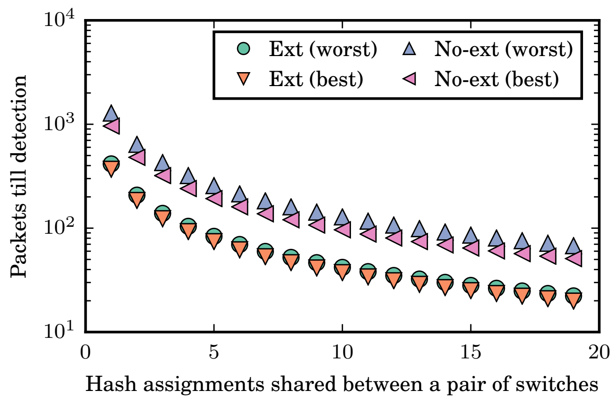
To understand if there is indeed a benefit, Fig. 12 compares the packets till detection when the servers are used, and not used in Preacher. Note that when the servers are used the total number of sampling nodes increases from 20 to 36. Detecting the malicious aggregate switch is the worst case, and detecting the malicious core switch is the best case. This is because the aggregate switch has the least number of pairs surrounding it, and the core switch has the most number of pairs surrounding it.

Controlling the Sampling Ratio. One of the benefits of using our approach is the flexibility offered in the type of packets that are sampled which is proportional to the number of packets sampled. For example, Preacher can configure the switches to sample only TCP handshake packets to monitor attacks on micro flows, or sample packets based on port numbers, e.g., UDP:53 for DNS, TCP:1812 for RADIUS authentication to monitor attacks on specific services. The sampling ratio can be controlled based on the time of day as well, e.g., sample TCP handshakes during peak activity and sample based on checksums during non-peak hours.

Extending Preacher to Other Networks. So far we have used a data center network (in particular, a Clos network) as our primary topology of Preacher. Of course, Preacher can also be employed in wide area networks (WAN) and ISP networks, within a single administrative domain. Indeed, we observe that ISPs are already moving towards a software-defined network [28]. Hence, Preacher can leverage the centralized control and global network in the ISP to detect forwarding attacks. In fact, we observe that using Preacher, one can obtain fine-grained monitoring over the network, as the sampling can be applied at switches (L2), and routers (L3) in unison.

In addition, we expect WANs and ISPs to have long paths (especially in terms of L2). Therefore, in the following we analyze the relationship between the detection probabilities, the path length and sampling ratios (number of hash assignments shared between a pair) using ATT's network topology. The ATT network we use, comprises of a total of 54 nodes with a median path length of 7 [14]. Recall the worst case is when the attacker is adjacent to the edge node. From Fig. 12 we can see that naturally, attacks can be detected sooner on long paths. However, as the path length increases from 7 to 10 for a fixed number of shared assignments, the number of packets till detection does not increase by much. If we look at the packets till detection for 10 hash values shared between a pair (530 sampling rules per node with a granularity $\delta = 1$), and the median path length of 7, we can see that on average an attack can be detected in less than 80 packets. 530 sampling rules per node in an ISP is arguably a small trade-off between improved security and reduced flow table space for forwarding in the router/switch.

Enriching and aggregating Preacher's incident reports. Fully utilized IDS systems usually provide the incident response teams (CSIRTs) a detailed and aggregated incident per attack. We observe that such enriched incidents can be provided by an independent mechanism that is fed with Preacher reports; we can aggregate them according to the localized attacking switch id and other parameters from the sampled packets, filter repeating events with the same root cause and



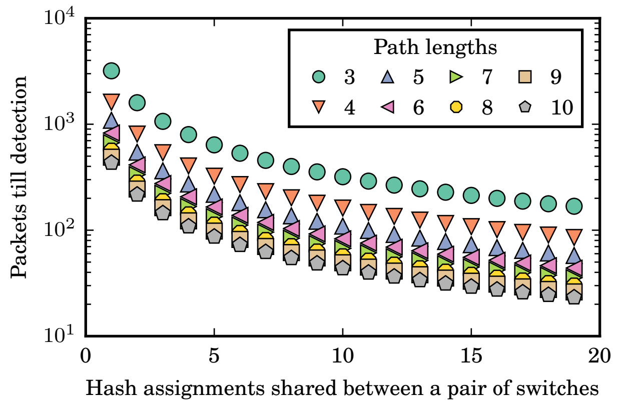


Fig. 12: Left: A theoretical comparison of the packets till detection when Preacher is extended to the edge (Ext) which involves 36 switches, and not (No-ext) which involves 20 switches. Right: A comparison of the packets till detection for the ATT network (54 routers) for different path lengths.

possibly perform more granular sampling at benign switches around the attacking switch. We leave such a mechanism as a future work.

B. Extending the Threat Model

In-band control plane threats. With in-band control plane, one switch may have access to the control (and sample) traffic of another switch. However, if the control traffic is an encrypted and authenticated TCP session we limit the attacker to only delay or drop packets (like in CrossPath Attack [29]), where dropping all packets for long time will result with TCP failure which can be detected. In order to detect (short time) packet delaying we suggest to deliver samples with timestamps.

As another layer of defense, which can also detect unauthorized communication within the in-band control plane, we can use sampling for the in-band control traffic as well. The expected increase of samples rate is insignificant considering a sum of geometric series with small constant.

Another way of one malicious switch to communicate with another remote switch is by a covert channel within an existing control session that traverses the remote switch on the way to the controller. This threat can exist in any in-band controlled network and in some cases it can be detected using the previous method (sampling control traffic). Specifically in cases where the covert channel resides within the reported samples (e.g., altering bits in the samples) then it may be detected when samples are correlated with other samples reported by other switches as part of the verification routine.

Large Fraction of Untrusted Switches. Preacher can support a large fraction of untrusted switches, even a majority, as long as attack locations along paths are surrounded by trusted switches. If untrusted switches can attack traffic before being sampled by trusted switches, or if there are no trusted switches at all, then detection might be avoided.

Such cases can be mitigated by extending the sampling to the end hosts as described previously (Sec. VI-A). This necessitates the use of keep-alives and an encrypted and authenticated channel (e.g., TLS) between each host and the controller to prevent eavesdropping and selective dropping of samples.

An alternative method to extend the security to the edge of the network could be to use a robust combiner approach [30], i.e., leverage physical redundancy (e.g., switches of different vendors or switches which were fabricated in different countries) or using different supply chains.

Covert Timing Channels. Although Preacher already supports detecting covert storage channels, detecting timing channels are currently not supported. However, with a time-based model of the network at the controller and accurate and precise sample timestamps, Preacher can detect such attacks. By comparing packet latencies from the expected samples with the time-based model, Preacher can uncover discrepancies and report them. For example, if the delay in receiving the sample is beyond the *timeout* then Preacher will throw two alerts: one drop attack alert for the first sample, and one inject attack alert for the second sample. It will not miss detecting such an attack.

Re-ordering and Low Volume Attacks. An attacker can reorder a transit packet thereby influence the order of the samples reported to the controller. Our system can be configured to also monitor inter-packet events, i.e., two packets sampled along the same path but in different order at different path locations. The probability to detect such events is lower than the other attacks considered in this paper, however, as with the other attacks, over time this probability increases. Additionally, if the re-ordering attack delays the packet by more than *timeout*, it can be discovered as the timing channels.

Preacher can probabilistically detect low volume network attacks such as TCP injection attacks [31]. In fact, Preacher can detect injection attacks that are on- and off-path, as our detection logic accounts for the source and destination addresses when comparing samples. Nevertheless, sampling *special interest packets*, e.g., TCP RST packets, can be achieved by configuring the switches with appropriate flow rules. Note that, the chance of detecting the off-path RST injection can be higher than detecting the on-path attack, as the attacker may have to guess the correct sequence and acknowledgment numbers and hence may send more than one RST packet.

VII. Related Work

Automated approaches to test and verify networks have received much attention over the last years, especially in the context of programmable networks and SDN [32], but e.g., also in the context of BGP [33] or MPLS [34]. Indeed, as discussed, SDN also provides the framework for our prototype implementation. In general, SDNs are known to introduce many flexibilities, also in terms of security, but they also introduce new threats [35], [36]. For example, the decoupling of the control plane in SDNs can introduce a risk of missed events and inconsistencies in the data plane, which led researchers to design approaches such as P4Consist [36] which actively generate traffic to verify whether the actual behavior on the data plane corresponds to the expected control plane behavior. However, these approaches are not designed to deal with malicious behaviors in the dataplane and can also be fairly inefficient if applied generally, due to the required large coverage of the dataplane state. Other examples dealing with security issues in SDNs are the distributed traffic monitoring system by Yu et al. [37] or FleXam [38], a sampling extension for monitoring and security applications in OpenFlow. NetSight [5] leverages SDNs to trace entire packet histories (without sampling), by collecting them “out-of-band”.

CherryPick [39] uses packets to carry information of SDN paths “in-band” (namely, a subset of links along the packet trajectory); however, these protocols struggle with drops and are not robust to malicious switches. In particular, the information CherryPick adds to the header along the path is only verified at the end of the path. Bates et al. [40] use SDN networks (plus some middleboxes) to observe the data plane behavior, even in the presence of malicious switches. Zeng et al. [41] use SDN to test the forwarding and policies in the network by generating and actively probing the data plane across the network. While Preacher is orthogonal to In-band Network Telemetry [42] (INT), Preacher is attractive for not requiring packet header modifications as needed in INT.

Shaghaghi et al. [43] propose WedgeTail, which adopts a symbolic representation of the network, and leverages Header-Space Analysis [44] to compute expected and actual trajectories. Preacher on the other hand can use a runtime global view of the network to construct requirements. Most importantly, we propose sampling packets based on the TCP/UDP checksum field, and comparing the samples at the collector, thereby detecting any modifications to a packet’s content. WedgeTail on the other hand relies on Netsight [5], or a packet header hash to receive packets, and cannot detect modifications to the packet’s content. Furthermore, WedgeTail is not meant for runtime detection.

Other works suggest traffic monitoring systems, such as WATCHERS [45] and Fatih [46], to detect misbehaved routers, however they require switch state of size proportional to the number of flows or path segments it monitors, and per packet state updates, thereby requiring also special switch designs. Our technique is supported in current switch implementations and is stateless.

WhiteRabbit [10], FOCES [9] use switch counters to detect forwarding attacks with small overheads however they can’t detect packet payload modification. Our technique use full packet samples and/or hashes across multiple locations therby supports packet payload modification.

Dinh et al. [47] propose to detect malicious switches by predicting the number of flows passing through switches periodically: going above or below the predicted value triggers an alarm. Predicting the flow count is however limited to detecting attacks that do not modify a packet header and/or payload, something Preacher can also do.

However, none of the approaches discussed above can be used to efficiently detect the adversarial attacks in the dataplane considered in this paper. In this regard, the paper most closely related to ours is by Lee et al. [14], who also study how to render sampling more secure in case of unreliable dataplanes, and who proposed a clever redundant sampling scheme, however, focusing only on drop and loop-based attacks. Our parallelizable detection algorithm in contrast is not limited to drops and does not require loops to manifest to detect injection based attacks. Furthermore, our detection algorithm does not rely on every node to report a key to identify the expected packet trajectory (traversal). Instead, we use the topology and policies available at the controller to compute the requirements of a sample. The detection algorithm proposed by us is on a per-packet granularity rather than an aggregation of trajectories and counters. We formally and empirically analyze our detection algorithm guarantees under various misbehaviors: denial-of-service, injections, mirroring, rerouting, collusion or modifications of headers and/or payloads. Moreover, we consider more generalized policies and also defend from attacks against the system components themselves.

VIII. Conclusion

In this paper we presented a highly-parallel and light-weight secure sampling approach that is designed for malicious environments. Our system, Preacher, can detect a wide range of misbehaviors (drops, inject, rerouting, header/payload modification, APTs, etc.), and in different settings, e.g., in datacenter or wide-area networks [48]. We implemented Preacher using OpenFlow to evaluate the detection time and throughput, and used analytical methods to explore the performance/resources tradoffs. We also evaluated the overhead introduced by Preacher, and identified a modest increase in resource utilization at the controller, but little to no overhead on the forwarding performance of the switch. This makes Preacher a promising network security tool for adversarial environments.

Looking forward, it will be interesting to study our approach in the context of stateful and more programmable data planes. On the one hand, this setting is more challenging, because state needs to be reported and verified as well. On the other hand, stateful data planes also introduce new opportunities, as it is now possible to perform certain tasks in the switches and hence reduce the load of the controller.

Acknowledgments The first author (K. T.) acknowledges the financial support by the Federal Ministry of Education and Research of Germany in the framework of the Software Campus 2.0 project nos. 01IS17052, the API Assistant activity of EIT Digital and the Helmholtz Research School in Security Technologies scholarship. The third author (S. S.) acknowledges the financial support by the Austrian Science Fund (FWF) project I 5025-N (DELTA). This work has further been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts, within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] K. Thimmaraju *et al.*, "Taking control of sdn-based cloud systems via the data plane," in *Proc. ACM Symposium on Software Defined Networking Research (SOSR)*, 2018.

[2] "Huawei HG8245 backdoor and remote access," https://websec.ca/publication/advisories/Huawei-web-backdoor-and-remote-access, 2013, accessed: 22-03-2021.

[3] "Snowden: The NSA planted backdoors in Cisco products," https://www.infoworld.com/article/2608141/snowden--the-nsa-planted-backdoors-in-cisco-products.html, 2014, accessed: 22-03-2021.

[4] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen, "Scion: Scalability, control, and isolation on next-generation networks," in *Proc. IEEE Symposium on Security and Privacy (SP)*, 2011, pp. 212–227.

[5] N. Handigol *et al.*, "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 71–85.

[6] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Trans. Networking (TON)*, vol. 9, no. 3, pp. 280–292, June 2001.

[7] V. Sekar *et al.*, "CSAMP: A System for Network-wide Flow Monitoring," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2008, pp. 233–246.

[8] Q. Li, X. Zou, Q. Huang, J. Zheng, and P. P. Lee, "Dynamic packet forwarding verification in sdn," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 6, pp. 915–929, 2018.

[9] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu, "Foces: Detecting forwarding anomalies in software defined networks," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 830–840.

[10] T. Shimizu, N. Kitagawa, K. Ohshima, and N. Yamai, "Whiterabbit: Scalable software-defined network data-plane verification method through time scheduling," *IEEE Access*, vol. 7, pp. 97 296–97 306, 2019.

[11] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*, 2015.

[12] S. Hong *et al.*, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*, 2015.

[13] S. Jero *et al.*, "Identifier binding attacks and defenses in software-defined networks," in *Proc. Usenix Security Symp.*, 2017.

[14] S. Lee, T. Wong, and H. S. Kim, "Secure split assignment trajectory sampling: A malicious router detection system," in *Proc. IEEE/IFIP Transactions on Dependable and Secure Computing (DSN)*, 2006, pp. 333–342.

[15] Federal Offic for Information Security (BSI), "The State of IT Security in Germany 2015," https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Securitysituation/IT-Security-Situation-in-Germany-2015.pdf, BSI, Tech. Rep., 2015, accessed: 9-03-2017.

[16] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of tor," in *Proc. IEEE Security & Privacy (S&P)*, May 2005, pp. 183–195.

[17] *OpenFlow Switch Specification*, Open Networking Foundation, 2013, version 1.3.2 Wire Protocol 0x04.

[18] Y. Afek, A. Bremler-Barr, S. L. Feibish, and L. Schiff, "Detecting heavy flows in the sdn match and action model," in *Computer Networks*, vol. 136, 2018, pp. 1–12.

[19] A. Bremler-Barr, D. Hay, I. Moyal, and L. Schiff, "Load balancing memcached traffic using software defined networking," in *Proc. IFIP Networking Conference*, 2017, pp. 1–9.

[20] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Passive and Active Measurement (PAM)*. Springer, 2015, pp. 347–359.

[21] P. Bosshart, D. Daly, and G. e. a. Gibb, "P4: Programming protocol-independent packet processors," *ACM Computer Communication Review (CCR)*, pp. 87–95, 2014.

[22] A. Bremler-Barr, D. Hay, I. Moyal, and L. Schiff, "Load balancing memcached traffic using software defined networking," in *Proc. IFIP Networking Conference*, June 2017, pp. 1–9.

[23] B. Heller. (2013) RipL: a Python library to simplify the creation of data center code. https://github.com/brandonheller/ripl. Accessed: 9-03-2017.

[24] R. Pang *et al.*, "A first look at modern enterprise traffic," in *Proc. ACM Internet Measurement Conference*, 2005, pp. 2–2.

[25] C. Partridge, J. Hughes, and J. Stone, "Performance of checksums and crcs over real data," in *ACM Computer Communication Review (CCR)*, vol. 25, no. 4. ACM, October 1995, pp. 68–76.

[26] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *SIGCOMM*, 2015.

[27] A. Gospodarek, "The Rise of SmartNICs – offloading dataplane traffic to...software," https://www.slideshare.net/LF_OpenvSwitch/lfovs17the-birth-of-smartnics-offloading-dataplane-traffic-tosoftware, 2017.

[28] S. Marek, "ATT's Stephens: 47% of Network Functions Are Virtualized," https://www.sdxcentral.com/articles/news/atts-stephens-47-network-functions-virtualized/2017/07/, 2017, accessed: 28-07-2017.

[29] J. Cao, Q. Li, R. Xie, K. Sun, G. Gu, M. Xu, and Y. Yang, "The crosspath attack: Disrupting the {SDN} control channel via shared links," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 19–36.

[30] A. Feldmann *et al.*, "Netco: Reliable routing with unreliable routers," in *IEEE Workshop on Dependability Issues on SDN and NFV*, 2016.

[31] N. Weaver, R. Sommer, and V. Paxson, "Detecting forged tcp reset packets." in *Proc. Internet Society Symposium on Network and Distributed System Security (NDSS)*, 2009.

[32] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic Foundations for Networks," in *POPL*, 2014.

[33] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 155–168.

[34] J. S. Jensen, T. B. Krogh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-rex: Fast verification of mpls networks with multiple link failures," in *14th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[35] K. Thimmaraju, L. Schiff, and S. Schmid, "Outsmarting network security with sdn teleportation," in *Proc. IEEE European Security & Privacy (S&P)*, 2017.

[36] A. Shukla, S. Fathalli, T. Zinner, A. Hecker, and S. Schmid, "P4consist: Toward consistent p4 sdns," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, pp. 1293–1307, 2020.

[37] Y. Yu, C. Qian, and X. Li, "Distributed and Collaborative Traffic Monitoring in Software Defined Networks," in *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014, pp. 85–90.

[38] S. Shirali-Shahreza and Y. Ganjali, "FleXam: Flexible Sampling Extension for Monitoring and Security Applications in Openflow," in *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013, pp. 167–168.

[39] P. Tammana, R. Agarwal, and M. Lee, "CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks," in *Proc. ACM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[40] A. Bates *et al.*, "Let SDN Be Your Eyes: Secure Forensics in Data Center Networks," in *Proc. NDSS Workshop on Security of Emerging Network Technologies (SENT'14)*, February 2014.

[41] H. Zeng *et al.*, "Automatic test packet generation," in *Proc. ACM CoNEXT*, 2012, pp. 241–252.

[42] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie, "In-band network telemetry (int)," *P4 consortium*, 2015.

[43] A. Shaghaghi, M. A. Kaafar, and S. Jha, "Wedgetail: An intrusion prevention system for the data plane of software defined networks," in *Proc. ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017, pp. 849–861.

[44] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 113–126.

[45] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson, "Detecting disruptive routers: A distributed network monitoring approach," *Network, IEEE*, vol. 12, pp. 50 – 60, 10 1998.

[46] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage, "Fatih: Detecting and isolating malicious routers," in *Proc. International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 538–547.

[47] P. T. Dinh, T. Lee, T. N. Canh, S. P. Dang, S. chul Noh, and M. Park, "Abnormal sdn switches detection based on chaotic analysis of network traffic," in *2019 25th Asia-Pacific Conference on Communications (APCC)*. IEEE, 2019, pp. 250–255.

[48] A. Gupta and et al., "SDX: A Software Defined Internet Exchange," in *Proc. ACM SIGCOMM*, 2014, pp. 551–562.

Kashyap Thimmaraju is a network security researcher. He received his PhD (2020) from Technische Universität Berlin and his MSc (2015) from UC Davis, California, USA. Previous to that, he worked for Juniper Networks in India and USA. His research interests lie at the intersection of networking, security, performance and systems.

Liron Schiff is Chief Scientist at Guardicore. For over a decade, Liron has been leading R&D projects in the cyber security industry along with academic research in the area of computer networks. His research focuses on programmability, resiliency and security aspects of networks. Liron holds a Ph.D in Computer Science from the Tel Aviv University, Israel, awarded in 2017 and was a visiting researcher at TU-Berlin, Germany and at the University of Vienna, Austria.

Stefan Schmid is a Professor at the University of Vienna, Austria. He received his MSc (2004) and PhD (2008) from ETH Zurich, Switzerland. Subsequently, Stefan Schmid worked as postdoc at TU Munich and the University of Paderborn (2009). From 2009 to 2015, he was a senior research scientist at the Telekom Innovations Laboratories (T-Labs) in Berlin, Germany, and from 2015 to 2018 an Associate Professor at Aalborg University, Denmark. His research interests revolve around the algorithmic problems of networked and distributed systems.