Preacher: Network Policy Checker for Adversarial Environments

Kashyap Thimmaraju
Technische Universitat Berlin, Germany
kashyap.thimmaraju@sect.tu-berlin.de

Liron Schiff
Guardicore Labs, Israel
liron.schiff@guardicore.com

Stefan Schmid
Facutly of Computer Science
University of Vienna, Austria
stefan_schmid@univie.ac.at

***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 adversarial switches or routers trying to reroute, mirror, drop, inject, or modify packets (i.e., header and/or payload) even under collusion. Additionally, 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.

***Keywords*-network security; programmable networks; SDN; data plane security; ONOS; OpenFlow;**

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].

While the problem of providing trustworthy networking with untrusted infrastructure is fundamental, our understanding of the solution space is limited. Incidents such as the recent Bloomberg reports on the SuperMicro hack [4], although currently unsubstantiated, forces us to ask ourselves what the consequences would be for networking equipment.

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 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, 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 good 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 insufficient in non-cooperative environments. A compromised switch/router can report falsified information.

Hence, we need a solution that can detect a broad spectrum of attacks in the presence of an adversarial data plane.

The basic idea and example. To this end, we have developed Preacher[1], a probabilistic policy checking scheme, that can provably and at runtime, detect several types of attacks arising from a malicious data plane. Preacher leverages 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.

In order to ensure scalability, and since simply inspecting all packets is infeasible, Preacher builds upon ideas by Lee et al. [8] and selects only part of the traffic, making the detection

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

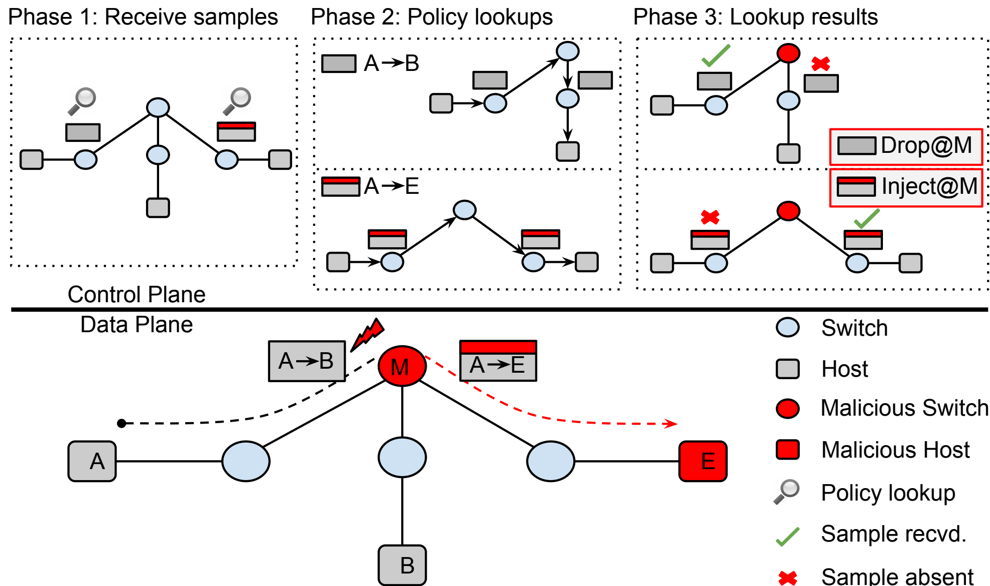

Fig. 1: Preacher detects an mitm (reroute) attack in 3 phases in the control plane. Host A sends a packet to B however, the malicious switch M, modifies the packet header (indicated by the red top) and sends it to E.

depend on the probability that attacked packets are inspected (in relevant locations).

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 release our entire framework as well as the collected data at https://www.github.com/securedataplane/preacher.

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 [9]–[11] on topology-based defenses.

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], [12], 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 [13] 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.

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).
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. The Preacher Scheme

Preacher securely and probabilistically samples packets in the data plane to discover the actions of malicious switches (as described in Sec. II) at runtime. Hence, we first provide

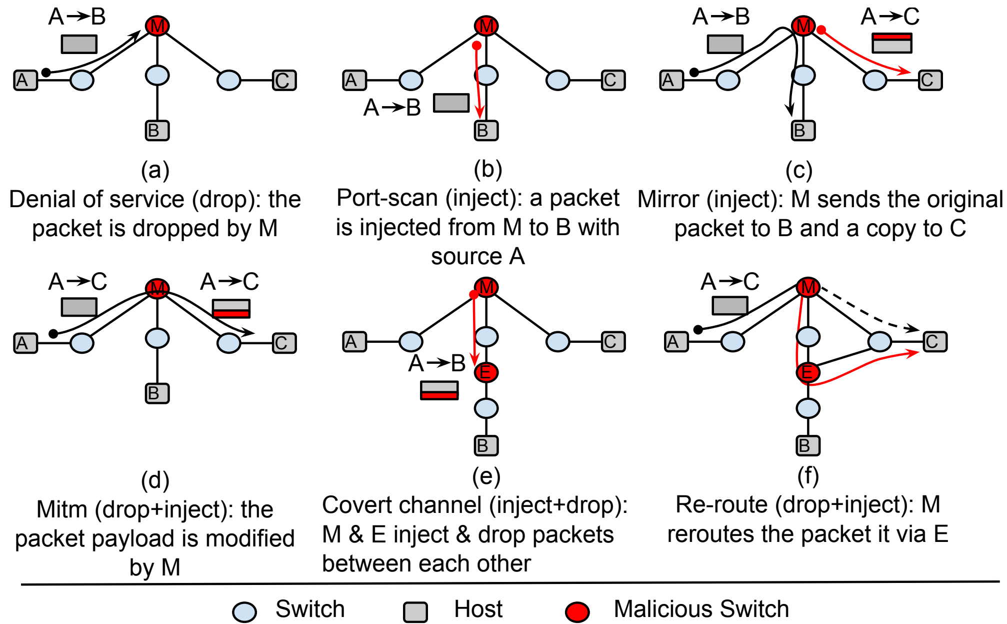

Fig. 2: A comprehensive overview of attacks, their respective primitives, and a short description of the attack.

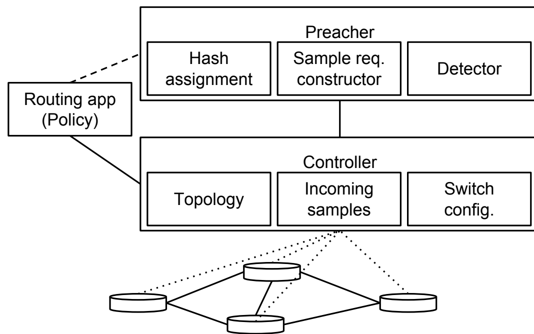

Fig. 3: Illustration of Preacher components and relation to existing network services.

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 more advanced cases 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, e.g, if samples are missing before or after the received sample, an injection or drop alert is reported resp.

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 on now.

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 [14] 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. [15] can be adopted. Alternatively, we can use P4 enabled switches to match the checksum fields or parts of the payload itself [16]. 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 [17]. 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 [8]. 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, 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 $History$.

Alg. 1 Detection

```
Require: assignments Assignment, switches S, samples queue Q, requirement
    policy Policy
 1: History ← ()                                              ▷ empty list
 2: t_0 = time()                                              ▷ current time
 3: while true do
 4:     timestamp, smp ← 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' ← History.get_min()
10:         traversal_set, ord ← Policy.get_possible_samples(smp')
11:         smp_set ← Assignment.remove_unassigned(traversal_set)
12:         if ∃smp1, smp2 ∈ smp_set: smp1 <_ord smp2 ∧ smp1 ∈
    History ∧ smp2 ∉ History then
13:             Report Drop of smp'.pkt between (smp1.s, smp2.s)
14:         if ∃smp1, smp2 ∈ smp_set: smp1 <_ord smp2 ∧ smp1 ∉
    History ∧ smp2 ∈ History then
15:             Report Injection of smp'.pkt between (smp1.s, smp2.s)
16:         for smp ∈ smp_set do
17:             History.remove(smp)
```

- *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 do. 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.

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. That 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 guaranties.

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 the technical report [18].

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 probability of detecting several attacks: (1) single packet attacks (drop and inject), (2) dropping an entire flow, (3) injecting a new flow, and (4) collusion.

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

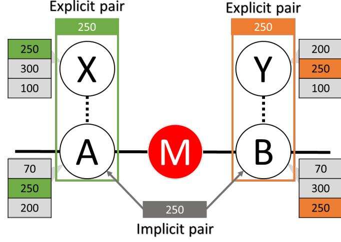

Fig. 4: Illustration of explicit and implicit pairs in a network with a pairwise assignment. The attacker's (M) position is (1,1). Explicit assignments for pairs $\{X, A\}$ and $\{Y, B\}$, indicated by the green and orange rectangles respectively, have the same value 250, making them an implicit assignment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

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 $q := \frac{p}{n-1}$ 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 doesn't depend on the network size (n), however the maximum sampling probability (p) that can be practically handled is usually inversely proportional to network size. We discuss resources vs. detection in Section IV-E.

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.

B. Flow Drop Attack

When the attacker drops an entire flow, clearly the detection probability depends on the number of switches traversed before and after the attack (k_1, k_2). By hashing the packet's payload, we obtain random per packet values. Hence, the detection of each packet in the flow is (ideally) an independent trial, each with a success probability P_{pa}. Therefore, the detection probability of the entire attack describes a geometric distribution. The expected number of dropped packets till detection is $1/P_{pa}$. The detection time is then the product of the average flow rate and $1/P_{pa}$.

Contrastingly, we observe that hashing only the packet header results in the same values for all packets in the flow. Hence, there is a constant detection probability P_{pa} (as for a single packet attack), regardless of the number of dropped packets.

C. Flow Injection Attack

Assuming that the injected flow's packets have uniformly distributed hash values (as assumed for the original flow), the detection probability is the same as the flow drop attack. The expected number of injected packets till detection is $1/P_{pa}$.

However, if all the attacker's packets hash to the same value, then the attack can either be detected from the first packet of the injected 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 injected flow. We assume that updated pairs are chosen following a random permutation of all $\binom{n}{2}$ pairs. We get that the expected detection time is approximately the maximum between $\binom{n}{2}/\lambda_u p$ and $1/\lambda_a p$ where p is the single packet attack probability, λ_u is the update rate (pair assignments per second) and λ_a is the attack rate (injected packets per second).

D. Collusion

Our analysis generalizes to collusion. Consider the scenario where one switch wishes to exfiltrate information to several other switches within the network as part of an APT attack. In the worst case, the colluding switches all know each others' assignments, and they do not report samples injected by any one of them. To analyze such behavior, we can adapt our above analysis. We compute the detection probability one attacker at a time, ignoring the other attackers along the path, effectively reducing the path length for the attacker under analysis. The final detection probability equals the probability of detecting at least one of the attackers.

E. Resources vs. Detection Time Tradeoffs

By applying the expected detection times to different Clos network sizes and hash assignments (pairwise and independent), we show the connection between the compute resources used by Preacher, in terms of inspected sample rate, and its performance, in terms of detection time. For this analysis, we consider a malicious core switch dropping (or modifying) all packets between two datacenter hosts.

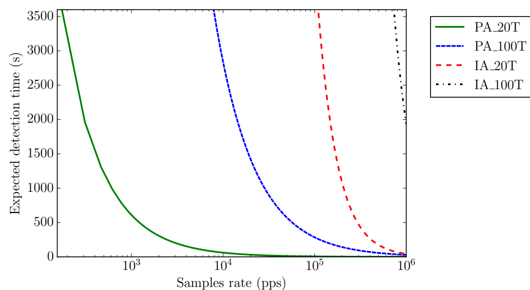

Fig. 5: Detection resources, in terms of inspected sampling rate, compared to detection time. Considering different Clos network sizes (number of ToR switches indicated by $\#T$), hash assignments pairwise (PA) or independent (IA), and varying the sampling probability (not shown on the graph). The following attack is analyzed: a core switch dropping (or modifying) all packets between two datacenter hosts.

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 [19]. Moreover we assume 20 hosts per rack and consider average and uniform inter-host traffic for the attacked packet rates.

Considering Preacher throughput of 1000 samples per second per core (see our methodology in Section V-D), we can see in Figure 5 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.

V. Experimental Evaluation

Using our prototype and realistic traffic workloads, we have conducted extensive experiments to evaluate Preacher. In particular, we evaluated the performance of our detection algorithm, investigating both the detection time as well as the detection throughput of Preacher. 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 as a Software-Defined Network (SDN). Indeed, SDNs 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 [20] 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.

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 [15] or P4 support and program customized packet parsers [21].

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 [22]. For realistic network traffic, we replay LLBNL traffic [23] 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. 6a 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 [24]. 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. 6b 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 see that experimentally detecting the two and three attackers takes longer than theoretically expected. Since the malicious switches do not attack packets they sample in the experiment, the detection takes longer. However, 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.

VI. Discussion

We now briefly discuss how Preacher can be improved further as well as how to handle special security cases. For a more detailed description of the suggestions, see our technical report [18].

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. We suggest to extend Preacher to other devices such as the hosts of the network and analyze the performance gain of such approach.

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.

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.

Detecting 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.

Detecting Re-ordering and Low Volume Attacks. Preacher can be configured to also monitor inter-packet events,

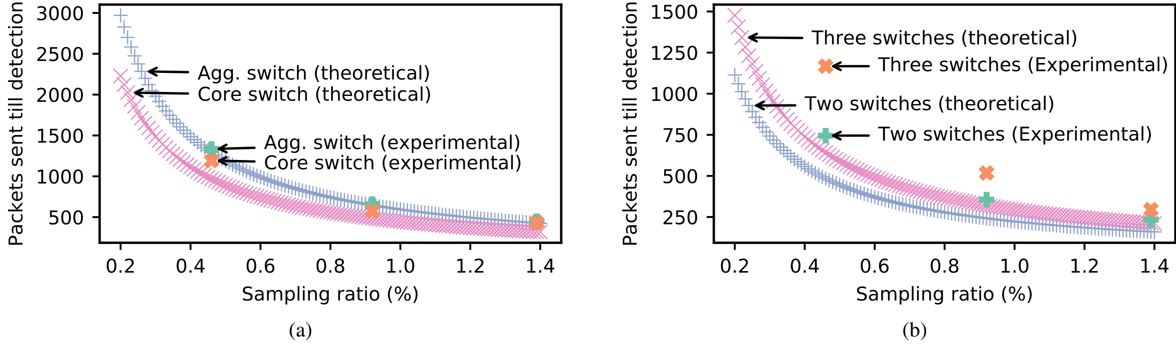

Fig. 6: (a) Average no. of packets to detect the flow inject attack at different positions using the pairwise assignment in the static setting. (b) Average no. of packets to detect the flow inject attack when two and three switches collude.

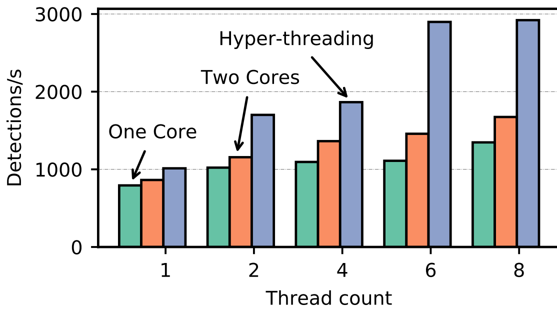

Fig. 7: Comparison of Preacher detection throughput with varying thread counts and CPU cores.

i.e., two packets sampled along the same path but in different order at different path locations. In addition we can detect low volume 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.

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. 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 [25]. Yu et al. [26] presented a distributed traffic monitoring scheme for SDNs, and FleXam [27] is a sampling extension for monitoring and security applications in OpenFlow. NetSight [5] leverages SDN to trace entire packet histories (without sampling), by collecting them "out-of-band".

CherryPick [28] 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. [29] use SDN networks (plus some middleboxes) to observe the data plane behavior, even in the presence of malicious switches. Zeng et al. [30] 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 [31] (INT), Preacher is attractive for not requiring packet header modifications as needed in INT.

Shaghaghi et al. [32] propose WedgeTail, which adopts a symbolic representation of the network, and leverages Header-Space Analysis [33] 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 [34] and Fatih [35], 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 by today switch implementations and is stateless.

The paper most closely related to ours is by Lee et al. [8], who also study how to render sampling more secure in case of unreliable dataplanes. However, their system focussed on drop and loop-based attacks. Our parallelizable detection algorithm is not limited to drops and does not require loops to manifest to detect injection based attacks. Moreover, 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: DoS, injections, mirroring, rerouting, collusion or modifications of headers and/or payloads. We consider more generalized policies and also defend from attacks against the system components themselves.

VIII. Conclusion

In this paper we presented a simple, 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 [36]. We implemented Preacher using OpenFlow to evaluate the detection time, and detection throughput. 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.

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 and 01IS1705, the API Assistant activity of EIT Digital and the Helmholtz Research School in Security Technologies scholarship.

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,” http://websec.ca/advisories/view/Huawei-web-backdoor-and-remote-access, 2013, accessed: 9-03-2017.

[3] “Snowden: The NSA planted backdoors in Cisco products,” http://www.infoworld.com/article/2608141/internet-privacy/snowden--the-nsa-planted\\-backdoors-in-cisco-products.html, 2014, accessed: 9-03-2017.

[4] J. Robertson and M. Riley, “The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies,” https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies, 2018, accessed: 5-11-2018.

[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] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] K. Thimmaraju, L. Schiff, and S. Schmid, “Preacher: Network policy checker for adversarial environments,” Technical Report, May 2019, ”www.dropbox.com/s/ywon20qxntgfv0q/Preacher_TR.pdf”.

[19] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *SIGCOMM*, 2015.

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

[21] 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.

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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