WedgeTail 2.0: An Intrusion Prevention System for the Data Plane of Software Defined Networks

Arash Shaghaghi
UNSW Sydney
7 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Mohamed Ali Kaafar
The Commonwealth Scientific and Industrial …
130 PUBLICATIONS **1,166** CITATIONS
SEE PROFILE

S. Jha
UNSW Sydney
257 PUBLICATIONS **3,617** CITATIONS
SEE PROFILE

WedgeTail 2.0: An Intrusion Prevention System for the Data Plane of Software Defined Networks

Arash Shaghaghi[1,2], Mohamed Ali Kaafar[2] and Sanjay Jha[1]

[1]School of Computer Science and Engineering, The University of New South Wales (UNSW), Australia
{a.shaghaghi, sanjay.jha}@unsw.edu.au
[2]Data61, CSIRO, Australia
{dali.kaafar}@data61.csiro.au

Abstract

Networks are vulnerable to disruptions caused by malicious forwarding devices. The situation is likely to worsen in Software Defined Networks (SDNs) with the incompatibility of existing solutions, use of programmable soft switches and the potential of bringing down an entire network through compromised forwarding devices.

In this paper, we present WedgeTail 2.0, an Intrusion Prevention System (IPS) designed to secure the data plane of SDNs. Our solution is capable of localizing malicious forwarding devices and can distinguish between the specicifc malicious actions of a compromised device such as packet drop, fabrication and modification. WedgeTail has no reliance on pre-defined rules by an administrator for its detection and may be easily imported into SDNs with different setups, forwarding devices, and controllers. The process begins by mapping forwarding devices as points within a geometric space and storing the path packets take when traversing the network as trajectories. Before inspection, the forwarding devices are clustered into groups of varying priority based on the frequency of occurrence in packet trajectories over specified time periods. The detection phase consists of computing the expected and actual trajectories of packets for each of the forwarding devices and 'hunting' for those not processing packets as expected. We have evaluated WedgeTail 2.0 in simulated environments, and it has been capable of detecting and responding to all implanted malicious forwarding devices within approximately an hour time frame over a large network. We report on the design, implementation, and evaluation of WedgeTail 2.0 in this manuscript.

Keywords: Software Defined Networks, SDN Security, Data Plane Security, Intrusion Prevention System

1. Introduction

Following the 'Vault 7: CIA Hacking Tools' revelations by WikiLeaks [12], Cisco confirmed that 318 models of its routers may be fully compromised through a simple command that may be invoked remotely [10]. In 2014, Edward Snowden also uncovered massive investments by NSA targeting core infrastructure and systems [7, 8]. At the same time, device vendors have been reported to leave backdoors in their devices as well (e.g. [4, 5]). Yet, the attack surface against forwarding devices is not limited to resourceful adversaries. Software and hardware vulnerabilities of the devices [9, 14, 18, 40] and vulnerable implementations of network protocols enable attackers to compromise forwarding devices. For instance, as reported in CVE-2014-9295 [11], a novice hacker could execute arbitrary code on routers simply through crafted packets targeting a specific function of the device [6].

A compromised forwarding devices may be used to drop or slow down, clone or deviate, inject or forge network traffic to launch attacks targeting the network operator and its users. For instance, the documents disclosed under 'Vault 7' revelations indicate that compromised routers may have been used for activities such as data collection, exfiltration (e.g. Operation Aurora [3]), manipulation and modification (insertion of HTML code on webpages) and cover tunnelling. A compromised routing system may be also used to bypass firewalls and intrusion prevention systems [23], violating isolation requirements in multi-tenant data centers [30], infiltrate VPNs [38] and more.

The emergence of Software Defined Network (SDN) paradigm brings promising opportunities to network management and security [13]. Nevertheless, SDNs are still vulnerable to data plane attacks and researchers have shown that compromised forwarding devices may even grant an attacker the capability to wrest control of an entire Software Defined Network [33, 35]. In this paper we look at the specific problem of protecting SDNs from malicious forwarding devices by determining if the traffic forwarding function of the switch itself is secure.

Securing the network against malicious switches has not been the subject of many studies in SDN security domain. The literature is mostly shaped by research towards developing secure controllers, novel SDN-enabled security services and real-time verification of network constraints (see [13, 52, 36] for surveys). However, no combination of these provides effective protection against compromised forwarding devices [21, 24, 17].

Recently, a few proposals aim to detect and mitigate threats associated with malicious forwarding devices. However, they are deemed as impractical suffering either from a simplistic threat model (e.g. [28]) or incurring substantial processing overhead to the network (e.g.[57, 58, 32]). For example, cryptographic solutions such as [32] have been designed to enforce path compliance in the presence of strong adversaries. Nevertheless, a universal deployment may be infeasible due to high overload required for per packet cryptographic operations and increased packet size.

SPHINX [21] is one of the solutions designed for securing the SDN data plane that does not assume forwarding devices are trusted. SPHINX detects and mitigates attacks originated from malicious forwarding devices through 1) abstracting the network operations with incremental flow graphs and 2) pre-defined security policies specified by its administrator. It also checks for flow consistency throughout a flow path using a similarity index metric, where this metric must be similar for 'good' switches on the path. We argue the following three factors as the main limitation of SPHINX. First, the system does not tolerate Byzantine forwarding faults. In other words, it does not assume malicious forwarding device may behave arbitrarily and therefore, SPHINX is not designed to detect the specific malicious actions performed such as packet drop, fabrication and delay. Second, the detection mechanism mainly relies on the policies defined by its administrator. In fact, the flow-graph component does not validate forwarding device actions against the controller policies but compared to their behavior through time. Hence, if the forwarding device(s) has been malicious since day one it will never be detected or when there are radical network configuration changes SPHINX will have large false positives. Third, SPHINX does not include a scanning regime and has no prioritization when inspecting the data plane for threats. Arguably, an important factor required for optimizing the detection performance in this context. Last but not least, SPHINX requires the majority of forwarding devices to be trustworthy. Although this assumption is realistic, a better solution solution will have to be independent of it.

Here, we introduce WedgeTail 2.0, a controller-agnostic Intrusion Prevention System (IPS) designed to 'hunt' for forwarding devices failing to process packets as expected. In essence, WedgeTail 2.0 is an extension to our earlier work [53] with improved scanning method, attack detection algorithm and packet trajectory computation engine.

The process begins by mapping forwarding devices as points within a geometric space and regarding packets as 'random walkers' among them. WedgeTail[1] tracks packet paths when traversing the network and generates their corresponding trajectories. At this point, in order to detect malicious forwarding devices, locate them and identify the specific malicious actions (e.g. packet drop, fabrication, etc.), it compares the actual packet trajectories with the expected ones (i.e. those allowed according to the network policies). If and when a malicious forwarding device is detected, WedgeTail responds as per the administrator-defined policies. For example, an instant isolation policy may be composed of two actions. First, the potentially malicious device is instructed to reset all the flow rules and then, it is re-inspected at various intervals by re-iterating the same packet(s) originally raising suspicion. If the malicious behaviour is persistent, the forwarding device may be isolated from the network.

In order to increase the probability of finding malicious devices, WedgeTail begins by prioritizing forwarding devices for inspection. In WedgeTail 1.0, we adopted Unsupervised Trajectory Sampling [48] to cluster forwarding devices into groups of varying priority based on the cumulative frequency of occurrence in packet trajectories - i.e. all of the trajectory database was analyzed. We improve the scanning mechanism in WedgeTail by adopting Time Period-based Most Frequent Path (TPMFP) technique [41] for analyzing the trajectories. The latter grants an administrator the capability to custom define the scanning periods as granular as the last few hours, weeks or months. For instance, an administrator may instruct WedgeTail to begin inspection from forwarding devices that processed most packets since the last major network change introduced a week ago. The latter will ensure prioritized inspection of critical forwarding devices potentially reducing the associated risks.

Wedgetail intercepts OpenFlow messages exchanged between the control and data plane and maintains a virtual replica of the network. It uses this virtual replica to compute the expected packet trajectories removing any trust on forwarding devices for this. Compared to WedgeTail 1.0, it improves the efficiency and performance of expected packet trajectory computation by replacing Header Space Analysis (HSA) [30] with NetPlumber [29], which is essentially an optimized and incremental version of HSA. On the other hand, in order to compute the actual packet trajectories, WedgeTail relies on NetSight [25] and queries for the packet history as it traverses the network. In fact, given the immense capabilities of NetSight for network troubleshooting, we expect this to be available as default in most SDNs. However, when NetSight is not available (e.g. in small-sized networks), the packets may be tracked using our custom packet tracking mechanism (see §6).

The rest of this manuscript is organized as follows. We begin with a succinct review of background information in §2. In §3, we discuss the main factors exacerbating the protection of SDN against malicious forwarding devices and outline the essential requirements of an effective solution. We define our threat model in 4 and provide an overview of the proposed solution in §5. WedgeTail's Detection Engine is throughly discussed in §6 detailing the scanning mechanism, actual and expected packet trajectory computation, and detection algorithms. WedgeTail's

[1]From hereon 'WedgeTail' indicates 'WedgeTail 2.0' and 'WedgeTail 1.0' indicates our earlier work [53].

Response Engine is presented in §7, followed by implementation and evaluation in §8 and §9, respectively. We conclude our work with a discussion on the importance of our work, comparison with related work and current limitations in §10.

2. Background

2.1. Header Space Analysis (HSA)

Header Space Analysis (HSA) [30] is a uniform and protocol agnostic model for debugging network configuration. HSA deals with a L-bit packet header as L-dimensional space, and models all processes of routers and middle-boxes as transfer functions, which transform subspaces of the L-dimensional space to other subspaces. Therefore, by analyzing forwarding rules of the network, HSA can calculate all the possible paths that a packet traversing the network on a certain port may take.

2.2. NetPlumber

NetPlumber [29] uses Header Space Analysis (HSA) as its foundation. However, it is much faster than HSA because instead of re-running every transformation every time the network changes, it incrementally updates only those transformations affected by the change. The heart of NetPlumber is the plumbing graph (see Figure 4) which captures all the possible flow paths in the network and is used to compute reachability. We discuss the integration of NetPlumber into WedgeTail in §6.

2.3. NetSight

NetSight [25] is a network troubleshooting solution that allows SDN application to retrieve the packet history. *netshark* is an example of tools built over this platform, which enables users to define and execute filters on the entire history of packets. With this tool, a network operator can also view the complete list of packet's properties at each hop, such as input port, output port, and packet header values. In §8, we present how WedgeTail inter-operates with NetSight to retrieve the actual packet trajectories.

3. Problem Description and System Requirements

As mentioned in §1, securing SDN networks against malicious forwarding devices is challenging. In fact, similar to [21], we also argue that the problem of protecting networks and their hosts against malicious forwarding devices is exacerbated in SDN context. We believe this is due to the following five main reasons[2]:

First and foremost is the incompatibility of existing solutions to secure SDN. In fact, due to the removal of intelligence from the forwarding devices, the defense mechanisms used for traditional networks may no longer work. Our analysis matches with the authors of [21] arguing that to import the traditional defenses into SDN we would need a fundamental redesign of OpenFlow [43]. A requirement which we deem as to be impractical.

The second factor is the unverified and complete reliance of control plane on forwarding devices. SDN controllers rely on *PACKET_IN* messages for their view of the network, however

[2]The first three factors are extracted from [21] with some minor amendments and additions.

this is not authenticated or verified. A malicious forwarding device may send forged spoofed messages to subvert the controller view of the network – even with TLS authentication in place. The same vulnerability enables a compromised forwarding device the capability to overload the controller with requests launching a Denial of Service (DoS) attack.

Thirdly, securing programmable soft-switches such as Open vSwitches is more challenging compared to switches. The former run atop of end host servers, which are more susceptible to attacks with attack surface much larger than physical switches.

Our fourth argument is that the SDN security domain is a moving target with the protocols and standards undergoing constant change. For example, several controllers have already been proposed with varying specifications undergoing constant updates. Hence, relying on the capabilities of one would limit practicality of a solution on another. The same argument is also valid for the OpenFlow [2] standard.

Fifth, a number of proposals have been made to delegate more control to the SDN data plane [46]. In fact, adding some intelligence and authority to the data plane has performance advantages such as lower latency response to network events and improves the network's fault tolerance through continuation of basic network operations under failing controllers. Furthermore, well-standardized protocols such as for encryption, MAC learning and codec control message (CCM) exchanges also require some intelligence at data plane. However, this proposals revive some of the vulnerabilities of traditional networks under SDNs and enlarge the threat vector.

Considering the aforementioned factors and driven by the limitations of existing work, we posit the 'must-have' features of an effective solution against malicious forwarding devices to include:

- Detecting and preventing all the different attacks against forwarding devices is indeed not practical (different vendors, software, standards, etc.). Therefore, along with the efforts to patch and protect the devices, we need to investigate the main functionality of the devices (i.e. packet forwarding). In other words, a compromised device (irrespective of how penetrated) alters the packet routing when delivering an attack and the proposed solution must be able to detect and react to this.

- To be efficient and effective, the proposed solution must be able to distinguish between the specific malicious actions (e.g. packet drop, fabrication, delay, etc.) and localize maliciousness.

- To reduce the detection time, it must systematically and autonomously prioritize forwarding devices for inspection.

- The response against threats must be programmable allowing an administrator to customize the protection to match the higher level network policies and requirements such as Quality of Service.

- The data plane is a critical component of the network infrastructure and the proposed solution must not disrupt the network performance during its inspection and analysis stage.

4. Threat Model

We assume a resourceful adversary who may have taken full control over one, or all, of the forwarding devices. This is, in fact, the strongest possible adversary that may exist at the SDN data plane, which to the best of our knowledge is not considered in the related work. For example, [21, 29, 30, 31, 42] assume all, or the majority, of the forwarding devices to be trustworthy. Interestingly, we have noticed an imprecise definition of adversary leading to oversights in SPHINX [21]. For instance, authors discuss an attack exhausting the TCAM memory of a switch that will cause a switch dropping packets over a period of time. As devastating as this may be, this device can not be used to execute attacks requiring packet modification or misrouting. Here, we assume the following capabilities for the adversary:

- The attacker may drop, replay, misroute, delay and even generate packets (includes both packet modification and fabrication), in random or selective manner over all or part of the traffic.

The above capabilities grant the adversary the capability to launch attacks against the network hosts, other forwarding devices or the control plane. For example, executing a Denial of Service (DoS) attack against the control plane by replaying or spoofing *Packet_In* messages. Note that detecting packet reordering is currently out of scope. The latter has previously been studied in the literature [44] and the proposed solutions are complementary to WedgeTail.

We regard a forwarding device as 'malicious' when both of the following properties hold A) the device is not handling the network packets according to the rules specified by the control plane, and B) the maliciousness is cloaked from basic troubleshooting tools. For example, the malicious device 'correctly' responds to *ping* or *traceroute* probes while corrupting other packets.

Arguably, the above characteristics may also be witnessed with a misconfigured, or faulty, forwarding device as well. In fact, the differentiating factor between these is the underlying intentions and hardly their behavior or impact. Hence, for the purpose of this work, we expand the definition of a malicious forwarding device to encompass both faulty and misconfigured devices. This implies that the proposed solution may potentially be used to detect faulty and misconfigured forwarding devices which are functioning anomalously – see Section §10.

We make the following assumptions for WedgeTail to work:

1. The control plane and its defined policies are trustworthy and securely transferred to the data plane (e.g. using TLS protocol [15]). There is an increasing body of literature aiming to achieve this, see [52, 13] for surveys. In other words, with SDN, the policy definition and enforcement points are separated in networks [36] and here, we exclusively focus on the the Policy Enforcement Point (PEP). Hence, preventing incidents such as [26] caused by erroneous administrator defined policies is out of scope.

2. WedgeTail is designed to detect forwarding devices failing to execute their main function (i.e. forwarding packets) and limit their threat. It is not designed to protect the devices from being compromised or detecting attacks targeting them. Specifically, in this context 'prevention' refers to the automated triggering of pre-defined policies against the identified threats.

3. The forwarding devices may not lie about their identity – a similar assumption is also made in [21].

5. WedgeTail: An Overview

In order to secure the SDN data plane, we propose WedgeTail[3]. WedgeTail is a controller-agnostic Intrusion Prevention System (IPS) designed to 'hunt' for forwarding devices failing to

[3]Australia's largest bird of prey and frequently cited in Aboriginal Australian stories and legends.

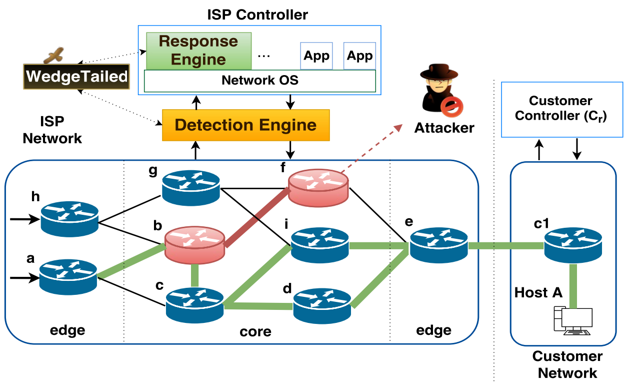


Figure 1: An abstract representation of WedgeTail over an ISP Network. The red devices are malicious and have been compromised by Attacker. The paths shown in green represent the expected paths for a packet send through forwarding device $fd(a)$ on port p_i to forwarding device $fd(c1)$ on port p_j at Customer Network. The path shown in red depicts a path the same packet took, which is not allowed as per the controller policies.

process packets as expected. As shown in Figure 1, it is composed of two main parts namely, Detection Engine (§6.3) and Response Engine (§7). The Detection Engine is responsible to retrieve the actual and expected packet trajectories, create the scanning regions and implement the attack detection algorithms. Accordingly, whenever a compromised device is detected, the Response Engine submits policies to the controller to protect the network.

5.1. How It Works?

Assume an ISP network as shown in Figure 1. WedgeTail retrieves 10010x10 ∪ 10011x10 as the header space of packets that may be sent over port p_i from forwarding device $fd(a)$ to port p_j of $fd(c1)$ using its integrated NetPlumber component. It also learns that the expected packet trajectories between these two nodes are as follows (shown in green colour in Figure 1): $fd(a) \rightarrow fd(b) \rightarrow fd(c) \rightarrow \{fd(d)\ OR\ fd(i)\} \rightarrow fd(e) \rightarrow fd(c1)$.

Malicious forwarding devices are detected as following: whenever the set of forwarding devices in the actual packet trajectories is not a subset of the expected packet trajectories then one or more of the forwarding devices in the actual packet trajectories may have been compromised – reminding that in §4 we extended the definition of 'malicious' to encompass both faulty and misconfigured devices. For instance, in Figure 1, the red colored trajectory is a non-allowed trajectory and $fd(b)$ is malicious.

Algorithm 1 presents WedgeTail's workflow. On each run, WedgeTail inspects the network on a specific port. The detection engine entails Find-Target-Forwarding-Devices() and Inspect-Device() functions in Algorithm 1. The former retrieves forwarding devices from the scanning regions and the latter, applies Algorithm 2 for each of the forwarding devices. If a forwarding device is detected as malicious, the Isolate-Forwarding-Device() function of Algorithm 1 along with the response policies specified by administrator is called.

6. The Detection Engine

We begin this section by providing definitions for Packet Network, Packet Path and Packet Trajectory in the context of our work. This is followed by a discussion on how WedgeTail com-

Algorithm 1 WedgeTail Detection and Response

```
Response Policy $RP$
Select $port_i \in \{\text{Port}\}$
**Find-Target-Forwarding-Devices** $(Port_i)$
Select $fd(i) \in \{\text{Target Forwarding Devices}\}$
**for all** $port_i \in \{\text{Port}\}$ **do**
    **Inspect-Device** $(fd(i))$
    **if** $fd(i)$ is ‘Malicious’ **then**
        **Isolate-Forwarding-Device** (RP, $fd(i)$)
    **end if**
**end for**
```

putes the actual and expected trajectories of packets. We then explain the scanning mechanism and conclude by presenting the attack detection algorithms.

Definition 6.1. (Packet Network). Simply referred to as ‘network’ in this manuscript, is a directed graph $G = (V, E)$ where V is a set of vertices representing forwarding devices and E is a set of edges representing links.

Definition 6.2. (Packet Path). Given G, an $fd(1) \rightarrow fd(j)$ path is a non-empty graph $P = (V_p, E_p)$ of the form $V_p = fd(1), fd(2), ..., fd(j)$ and $E_p = \{(fd(1), fd(2)), ..., (fd_(i), fd(j)\}$ such that P is a sub-graph of G and the $fd(x)$ are all distinct.

Definition 6.3. (Packet Trajectory). Given G and packet n, a Packet Trajectory P_nT is a sequence $((fd(1), t_1), (fd(2), t_2), ..., (fd(j), t_j))$ where t_x is a timestamp indicating the time when P_nT passes $fd(x)$ and there exists a path $fd(1) \rightarrow fd(2), \rightarrow, ..., \rightarrow fd(j)$ on G.

Simply put, a packet trajectory is the route a uniquely identifiable packet takes while traversing a network from one forwarding device to another. For each trajectory the times that the packet meets each forwarding device along its path is recorded along with a timestamp for the whole trajectory. For instance, the timestamp DD.MM.YY is attached to trajectory P_nT, which passes through nodes $fd(i)$ and $fd(j)$ at times t_i and t_j, respectively. We consider different paths for the same packet as distinctive trajectories. In other words, packets with the same header may be routed through different paths in respect to the network condition on each iteration. For instance, as shown in Figure 1, a packet traversing through the green line from $fd(a)$ to $fd(e)$ may be routed through $fd(i)$ or $fd(d)$ depending on the QoS requirements. Note that multiple repetitions of the same path for the same packet is only regarded as one trajectory.

6.1. Packet Trajectories

Actual Packet Trajectories: We propose two different solutions to retrieve the packet trajectories. As succinctly reviewed in §2, NetSight is a recently proposed network troubleshooting solution that allows retrieving all the forwarding devices that a packet visited while traversing the network. Therefore, if NetSight was deployed on a network, a convenient approach would be to query for packet history and create the trajectories. We achieve this by developing an integrated module that queries NetSight for packet histories using available APIs. This approach is the preferred method as is systematic and scales well.

An alternative approach would be for WedgeTail to run a deterministic hash function over a packet header and use this hash to track the packet as it traverses the network (i.e. generating

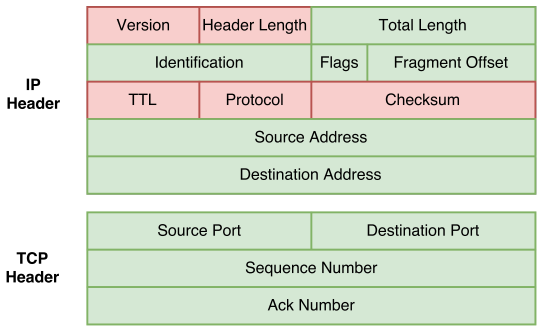


Figure 2: Packet header fields used for labeling.

labels). The choice of an appropriate hash function would be crucial for this matter as is selecting the proper packet header values. To achieve this, we use the packet hashing function used in [22]. We pick packet header values such as source address, destination address from IP header and source port and destination port in TCP header – the values used for the hashing are shown in Figure 3. Note that in practice the labels can be quite small (e.g., 20 bit) – although the size of the packet labels depends on the specific situation. Therefore, the overhead to collect trajectory samples is also small since the traffic that has to be collected from nodes only consists of such labels (plus some auxiliary information).

An issue to consider is the impact of collision in our proposed hashing-based mechanism. First, we envision our custom hashing to be used as an alternative where NetSight is less likely to be deployed such as in smaller networks. In other words, the the likelihood of collision decreases with smaller traffic volumes. Second, if collisions result in 'impossible' paths (e.g. $fd(a) \rightarrow fd(f)$, where there is no direct link between the two) then this will be ignored by WedgeTail. The only case that a collision may result in false positives is when it results in an invalid trajectory that has at least the first two forwarding devices same as a valid trajectory. For instance, WedgeTail falsely detects $fd(b)$ as malicious if the invalid trajectory is $fd(a) \rightarrow fd(b) \rightarrow fd(f) \rightarrow fd(e)$ and the valid trajectory is the green ones shown in Figure 1 (see §6.3 for attack detection algorithms).

Expected Packet Trajectories: WedgeTail uses NetPlumber to compute the expected packet trajectories. NetPlumber is run on a virtual replica of the network, which WedgeTail maintains by intercepting the OpenFlow messages exchanged between data and control plane (see §8 for details). The virtual replica enables computing the expected packet trajectories when traversing the network without trusting the forwarding devices, which may be malicious.

NetPlumber is initialized by examining the forwarding tables of virtual network replica to build its Plumbing Graph. This is then used to retrieve all the possible paths that a packet from port p_i of $fd(i)$ may take as it moves to port p_j of $fd(j)$.

To be current, NetPlumber updates its plumbing graph whenever a new rule, link or table is added or removed. For instance, when a new rule is added, NetPlumber first creates pipes from the new rule to all potential next hop rules, and from all potential previous hop rules to the new rule. It also needs to find all intra-table dependencies between the new rule and other rules within the same table. Next, NetPlumber updates the routing of the flows. To do so, it asks all the previous hop nodes to pass their flows on the newly created pipes. The propagation of these flows then continues normally through the plumbing graph.

The incremental approach used in NetPlumber makes it a much more efficient solution than

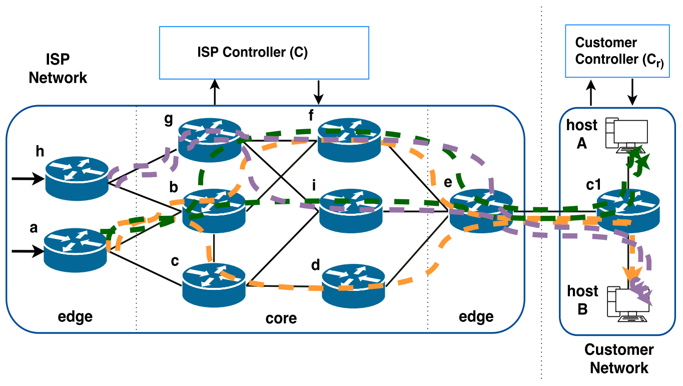


Figure 3: Forwarding devices in ISP network. The dotted lines represent packet trajectories.

HSA, which is also evident in our evaluation results (see §9). We refer the interested reader to [29] for more information on the functioning and performance metrics of NetPlumber.

6.2. Scanning Mechanism

WedgeTail prioritizes forwarding devices for its inspection. The core idea is that the analysis has to begin from the forwarding devices that the majority of packets encounter while traversing the network. To identify these, WedgeTail keeps track of trajectories for all packets on all ports over time and finds the most frequent paths over specified periods. For instance, assume the trajectories drawn in Figure 3 represent all the trajectories recorded for ISP network in the last 7 days. It is evident that $fd(b)$, $fd(g)$ and $fd(f)$ are more commonly encountered by packets. Indeed, identifying these is much more complicated in a large network with potentially massive number of trajectories. We adopt the solution proposed in [41] to process the associated large-scale trajectory database and extract the most frequent path over certain periods.

Processing trajectory datasets while focusing on their inherent spatiotemporal features has been extensively studied in the last decade. The most popular approach is shortest path finding [39], which does not fit our scope. Alternatively, solutions such as [51, 27] detect hot routes and trajectory patterns with heavy traffic. We used a similar mechanism known as Unsupervised Trajectory Sampling [48] in WedgeTail 1.0 to detect the denser regions and prioritize the related forwarding devices in inspection. However, such solutions retrieve patterns in a global manner with no consideration of time or specific source and destinations.

Most Frequent Path (MFP) is another way of extracting useful information from trajectory datasets. It is mainly used to reflect on the common routing preferences and has various usages and real world applications such as investigating people's travel habits, path recommendations, and etc. [41] is one of the very few solutions proposed to study the MFP problem with user-specified time periods. Moreover, it is designed for large trajectory datasets which makes it a perfect match for our scenario. The proposed two-step strategy first constructs a graph with the frequencies of the candidate paths and then executes a graph search to find the results. It proposes Footmark Index (FMI) and Containment-Based Footmark Index (CFMI) to reduce the number of random disk accesses by only fetching the dominant trajectories. To find the results, it proposes a new variant of the classic Bellman-Ford algorithm to deal with the sequence-valued path properties. We refer the interested reader to [41] for algorithm listing and further details.

Using time-period based most frequent path (TPMFP), WedgeTail reduces the trajectory dataset into a representative sample that encapsulates the most commonly visited forwarding devices over the specified time period. These forwarding devices are allocated a higher priority when WedgeTail begins its inspection of data plane forwarding devices.

6.3. Attack Detection

The main attack detection algorithm (*Inspect-Device()*) is presented in Algorithm 2. The algorithm takes as input a target forwarding device and a port and returns a malicious node specifying its malicious action. First, a snapshot of all network forwarding device configurations is retrieved. Accordingly, the trajectories that a packet may take against each of the other forwarding devices, and the control plane, is computed – note that the packets required for creating the trajectories are chosen randomly in the control plane and cannot be known by an attacker to influence this process. Thereafter, the actual trajectories for the selected packets are retrieved using mechanisms discussed in §6.1. At this point, whenever the set of forwarding devices in the actual trajectory is not a subset of the expected trajectories, a malicious forwarding device is detected.

Formally, let A denote the total ordered set of actual forwarding devices for a packet traversing from target $fd(i)$ to $fd(j)$ and E the ordered set of expected forwarding devices for the same trajectory. If A $\nsubseteq$ E then $fd(i)$ is malicious. The malicious actions are identified as follows[4]:

Algorithm 2 Attack Detection Algorithm

```
Inspect-Device($fd(i)$, $PortP_i$) {
Status S = Check-State-Change();
File F = Dataplane-Configurations-Snapshot(S);
while Check-State-Change() == S do
    List L = F.ForwardingDevices() – $fd(i)$
    for all $fd(j) \in$ L do
        Packet Pck;
        Trajectory Actual, Expected;
        Pck.Source() = $fd(i)$;
        Pck.Destination() = $fd(j)$;
        Pck = Find-Packet(Pck.Source, Pck.Destination);
        Expected = HSA-Trajectory(Pck);
        Actual = Actual-Trajectory(Pck);
        if Actual $\neq$ Expected then
            Identify-Attack($fd(i)$, Port(i));
        end if
    end for
end while
}
```

1. Packet Replay: Occurs when a forwarding device sends a copy of the packet to a third destination as well as the intended destination. Figure 1 shows a packet replication attack example, where $fd(b)$ replicates packets to $fd(f)$ which in turn an attacker may use to forward some,

[4]Note that to simplify the descriptions, without loss of generality we assume that there exists only one valid trajectory between two forwarding devices.

or all, of traffic to a machine under his control. A forwarding device that replays packets(s) enables an attacker to execute attacks such as surveillance attack and authentication attack.

Detection: Let $fd(k)$ be a forwarding device other than $fd(i)$ and $fd(j)$. A' be the set of forwarding devices in the actual path excluding $fd(k)$, or $A - \{fd(k)\}$. If $\exists fd(k) \in A : fd(k) \notin E$ and $A' \subseteq E$ then WedgeTail detects a packet replay attack.

2. Packet Misrouting: Occurs when a packet is diverged from the original destination and does not reach its intended destination. This may be used to launch an attack against network availability or as part of more complicated threats. For example, by forming a triangle routing and creating routing loop resulting in packet TTL value expiration the network congestion may result in a partial, or total, shutdown of the network.

Detection: Let $fd(k)$ be a forwarding device other than $\{fd(i), fd(j)\}$ and A' be the set of forwarding devices in the actual path excluding $fd(k)$, or $A - \{fd(k)\}$. If $\exists fd(k) \in A : fd(k) \notin E$ and $A' \nsubseteq E$ then WedgeTail detects a packet misrouting attack.

3. Packet Dropping: A compromised forwarding device that drops packets creates a black or grey hole in the network. In the former, it drops all the packets, and in the latter, it drops packets periodically or retransmission of packets or drops packets randomly. Packet dropping is used in attacks such Denial of Service (DoS) against network provider.

Detection: WedgeTail detects packet dropping if $A \nsubseteq E$ and card(A) < card(E).

4. Packet Generation: A compromised forwarding device may fabricate packets or modify existing ones. This may be used to mount attacks such as DoS. Such changes are detected by WedgeTail through its labeling mechanism. In other words, once any attribute used for labeling packets is changed, the label is changed, and the trajectory is undefined.

Detection: WedgeTail detects packet generation whenever $A \nsubseteq E$ and $E - A = E$.

5. Packet Delay: Occurs when a forwarding device delays the traffic and increases jitter. The delay of a TCP stream may causes spurious timeouts and unnecessary re-transmissions, which severely threatens the TCP throughput [59]. Packet delay is a serious threat against time-sensitive traffic [24].

Detection: Let T_e be the estimated time for $packet_i$ moving from $fd(i)$ and $fd(j)$ over a trajectory $\bar{\tau}$. Accordingly, let T_a be the actual time that it took for this packet to traverse $\bar{\tau}$. Assume the maximum valid delay due to network congestion on this trajectory is T_d. If $\Delta T_{e,a} > T_d$ then there is a packet delay attack.

Note that the estimated time may be set to be the average time for all packets traversing that route or computed by sending simple *ping* packets. The maximum valid congestion may be computed using [49] or [55], where it is possible to achieve real-time congestion detection and measurement.

6.4. Malicious Localization

As mentioned a trajectory is regarded as a total ordered set. Once one of the malicious actions are detected, it is possible to locate the associated forwarding device by comparing A and E (see previous section). Consider Figure 1 and assume when inspecting $fd(a)$ we retrieve $E(\bar{\tau})$ and $A(\bar{\tau})$ as expected and actual trajectories between $fd(a)$ and $fd(e)$, respectively.

$A(\bar{\tau})$: $fd(a) \rightarrow fd(b) \rightarrow fd(f) \rightarrow fd(e)$ equivalent to total ordered set $E = \{fd(b), fd(f)\}$.

$E(\bar{\tau})$: $fd(a) \rightarrow fd(b) \rightarrow fd(c) \rightarrow fd(d) \rightarrow fd(e)$ equivalent to total ordered set $A = \{fd(b), fd(c), fd(d)\}$

In this case, by intersecting E and A we retrieve that $\{fd(b)\}$ is the malicious node, where packet misrouting was initiated. The analysis is continued with $fd(c)$ and $fd(d)$ - i.e. $A - E$ -

so that at the end of this process any forwarding device that my be malicious is identified. The same approach can be used for malicious actions 1, 3 and 4. To locate a forwarding device that is delaying packets we retrace time hop by hop in A and compare with the expected time.

6.5. Practical Considerations

Network congestion will result in packet drops and delays. Therefore, to minimize the number of false positives, WedgeTail has to estimate with a high accuracy the number of packets drops and delays associated with network congestion. Several solutions have already been proposed in the literature to achieve this. Authors of [45] propose a solution to detect packet dropping or gray hole attacks in networks by exploiting the correlation between packet delays and packet losses due to congestion. Their proposed methodology is based on passive observations of the one-way network delay experienced. For the scope of this work, the main advantage of this solution compared with the better-known proposals such as [45] is that we could implement it without any additional overhead or support from the network. For instance, [45] assumes the routers in the network provide real-time data related to the queue lengths at their interfaces.

6.6. Optimizing Scans

Compared to WedgeTail 1.0, we optimize attack detection by reducing the number of operations while iterating through the forwarding devices in Algorithm 2. We explain this using an example. Consider the case where WedgeTail is inspecting $fd(a)$ on Port P_i and the green link showing all the possible paths between $fd(a)$ and $fd(e)$. Referring to Algorithm 2, the process involves setting $fd(b), fd(c), fd(i)$ and $fd(e)$ as destinations and comparing the actual and expected trajectories in each case. However, there is repetition in this process, which may be skipped. For instance, evaluating $fd(a) \rightarrow fd(i)$ involves $fd(a) \rightarrow fd(b)$, $fd(b) \rightarrow fd(c)$ and $fd(a) \rightarrow fd(i)$. Hence, when evaluating $fd(a) \rightarrow fd(d)$, WedgeTail skips the repeated paths and only inspects $fd(c) \rightarrow fd(d)$.

7. The Response Engine

WedgeTail can be programmed to automatically reply to identified threats using its response engine. The response engine takes as input a set of XML-formatted policies and translates them into actions for the controller. Developing a fully fledged policy engine and ensuring the logical correctness of them is out of scope for this work. We developed a simplified policy engine for our initial evaluation of WedgeTail.

Policy Syntax: Each policy requires six main features and attributes describing them. The features include Subject, Object, Actions, Condition, Exception and Expiration time. Table 1 lists the attributes currently supported for these features. The naming used for attributes are assumed to be self-descriptive. Note that the values in parenthesis are expected to be provided as input for each of these attributes. While each policy may have only one subject, the other features may have more than just one associated attribute. The Exception attribute is mainly used to build hierarchy for the policies and Expiration is used to specify the validity period.

We now look at two examples. Consider Figure 1 and assume only $fd(f)$ is detected as malicious. An administrator-defined policy may specify two different policies matching this forwarding device (i.e. one using the Forwarding Device attribute and another using Controller attribute). First, it may specify $fd(g)$ as subject and instruct it to use an alternative route to forward traffic. Second, it may specify for the Controller to block all incoming OpenFlow messages.

Feature	Attributes
Subject	Forwarding Device(id) | Controller
Object	Packet(id) | Flow(id) | Switch(id)
Action	Isolate(fd(id)) | Update_forwarding_table(fd(id)) | Alarm | Block_Messages(fd(id)) | Test_Again(fd(id))
Exception	Policy P_i
Expiration	T (millisecond)

Table 1: Overview of the response engine policy syntax

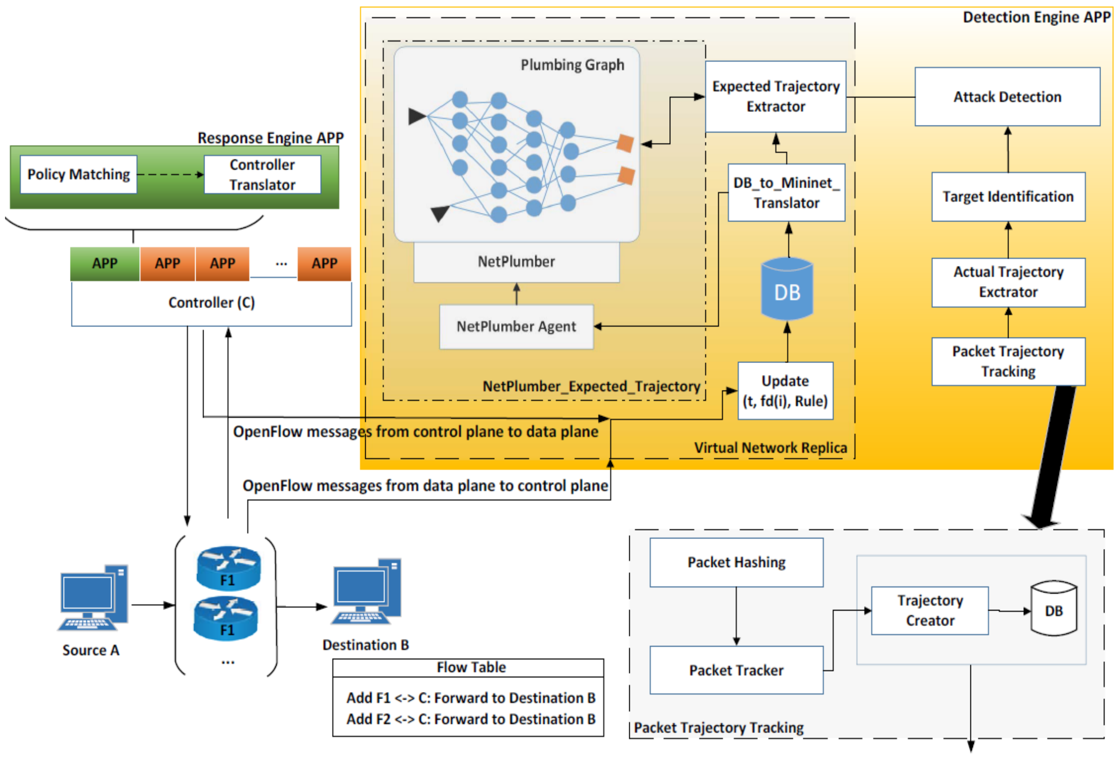
Now, consider the same scenario as before but this time with only $fd(b)$ identified as malicious. In this case, there may be an Exception feature stating if a policy for $fd(f)$ is still active then no action is executed from this policy.

8. Implementation

We envision WedgeTail to be integrated as an application for SDN controllers for both detection and response. However, at this stage, to demonstrate WedgeTail's compatibility with different platforms and evaluate it over different controllers we implemented the detection engine as a proxy sitting in between the control and data plane. We report that a similar architecture is also used in [21]. Another reason for this design choice was that the detection engine requires advanced functions that is not consistently available across different controllers. Currently, the response engine is programmed as an application for Floodlight [**?**] controller.

WedgeTail's architecture is shown in Figure 4. We implemented our solution mostly in Java using approximately 2000 lines of code. WedgeTail begins by creating the scanning regions. To do this, it creates a unique hash from a large number of packets. The packets are then continuously tracked as they traverse the network by intercepting the $PACKET\_IN$ messages sent from the data plane to control plane. $Actual\_Trajectory\_Extractor$ generates the resulting trajectories and stores them in a trajectory database along with some packet information and timestamps.

Once the scanning zones are generated and the target forwarding devices are identified, WedgeTailed requires having the expected trajectories of packets to initiate its inspection. Hence, it queries the controller for current topology and launches a Mininet matching the same setup. It then intercepts all the OpenFlow messages exchanged between the control plane and data plane including $FLOW\_MOD$ and $PACKET\_IN$ messages. The OpenFlow messages sent from the controller to forwarding devices (e.g. $FLOW\_MOD$) is first translated into a database INSERT command. This command stores the rule, forwarding device receiving the rule along with a time value in a MySQL database. Thereafter, using the $DB\_to\_Mininet\_Translator$ component, these are retrieved from the database and translated into appropriate Mininet commands. The result is a virtual network replica, which is continuously updated. The virtual network replica is used by $NetPlumber\_Expected\_Trajectory$ module to compute the expected trajectories of packets. We have noticed NetPlumber performance degrading with major network configuration changes, therefore in such cases, NetPlumber is manually instructed to reset its Plumbing Graph.



Detection Engine APP
Plumbing Graph
Expected Trajectory Extractor
Attack Detection
DB_to_Mininet_ Translator
Target Identification
NetPlumber
NetPlumber Agent
DB
Actual Trajectory Exctrator
Packet Trajectory Tracking
NetPlumber_Expected_Trajectory
Update (t, fd(i), Rule)
Virtual Network Replica
Response Engine APP
Policy Matching
Controller Translator
APP
APP
APP
...
APP
Controller (C)
OpenFlow messages from control plane to data plane
OpenFlow messages from data plane to control plane
F1
F1
...
Source A
Destination B
Flow Table
Add F1 <-> C: Forward to Destination B
Add F2 <-> C: Forward to Destination B
Packet Hashing
Trajectory Creator
DB
Packet Tracker
Packet Trajectory Tracking

Figure 4: WedgeTail Architecture

9. Evaluation

We evaluated WedgeTail over simulated networks, which were different in terms of the number of forwarding devices, forwarding rules, network subnets, and trajectories – with our latest simulation closely resembling real-world network conditions. We replicated a number of attacks against SDN networks that were previously reported in the literature and analyzed the accuracy of our solution in detecting these attacks. In order to further evaluate WedgeTail's detection engine, we wrote scripts that synthetically implanted a total of 1500 attacks covering all of the malicious actions specified in §4 over our simulated networks.

Here, we report on WedgeTail's accuracy and performance including metrics such as detection and prevention success, average detection time, user perceived latencies and overheads related to policy verification. To resemble real world network conditions, we introduced congestion in our simulated network causing packet losses and measured the associated false alarms.

We compare our proposal with related works and argued how WedgeTail, in most cases, outperforms them both in detection and response. Furthermore, in order to analyze the impact of changes introduced with WedgeTail 2.0 we compare our performance metrics with that of WedgeTail 1.0 [53]. We also show WedgeTail's broad utility by illustrating how it can support disparate networking requirements over 2 different cases studies.

Finally, given that target identification and virtual network replica reconstruction are new features introduced as part of WedgeTail and may be of use in other domains, we report on their performance separately.

Number of	AARNet Setup	Zib54 Setup	Sprint Setup
Forwarding Device	12	54	316
Subnet	48	872	52,745
Rule	472	23,962	15,871,528
Trajectory	634	49,174	1,134,429

Table 2: Simulated networks compared in terms of total number of forwarding devices, subnets, rules and trajectories.

9.1. Experimental Setup

We simulated three different networks namely *AARNet*, *Zib54* and *Sprint*. AARNet setup was used in our initial feasibility study and resembles a minimalistic backbone ISP network topology with only 12 forwarding devices. The forwarding rules in this network reached 472 and we generated benign traffic such that about 600 trajectories were available in the trajectory database. In Zib54 Setup, we extended the network size and had 54 forwarding devices. The trajectory database used by WedgeTail contained about 24000 forwarding rules and 49000 trajectories over 870 subnets. A large network is presumed to have more forwarding devices as well as many more trajectories. Therefore, we evaluated WedgeTail on the Sprint Setup, which was a much larger network containing 316 forwarding devices with more than 1 million trajectories, 15 million rules and 52000 subnets.

Network Topologies: The network topologies for AARNet Setup, Zib54 and Sprint were extracted from The Internet Topology Zoo [34], SNDlib [47] and Rocketfuel [54], respectively. Figure 11.a, 11.b (in Appendix Section) represent AARNet Setup and Zib54. In these setups, each node is assumed to contain only one forwarding device, and there is only one link in-between these devices as also depicted in the figures. Figure 11.c (in Appendix Section) depicts the interconnection of different domains at Sprint backbone network, which we used as the network topology for Sprint Setup. Note that in Figure 11.c, for clarity, the forwarding devices at each node are not depicted, and only one link connects the nodes to each other.

Flow Entries: We are unaware of any publicly available flow entry data set for our simulated networks. Hence, to add flow-entries, we created an interface for a subset of prefix found in a full BGP table from Route Views [50] and spread them randomly and uniformly to each router as 'local prefixes'. We then computed forwarding tables using shortest path routing. The resulting forwarding rules and subnets for each setup are shown in Table I. We report that a similar methodology is also adopted by related work such as [17], [56].

Traffic Generation: We used Mausezahn [1] and a custom script to add benign traffic to the networks. Similar to [21], our custom written script imported three real-world network traces from [19, 20, 37] to drive traffic into Mininet.

We hosted the simulated networks on a machine equipped with Intel Core i7, 2.66 GHz quad-core CPU and 16 GB of RAM. The SDN controller equipped with WedgeTail was also hosted on a machine with the same specifications.

9.2. Attack Scenarios

We revise the main characteristics of six different attack scenarios originating from the data plane of Software-Defined Networks. Thereafter, we discuss how WedgeTail detects them in our evaluations and compare the advantages of WedgeTail over SPHINX in terms of detection methodology. Note that the authors of [21] did not provide the code of their solution on request and therefore, we cannot provide a numerical performance comparison at this time.

The following attack scenarios were evaluated over networks equipped with OpenDaylight (ODL), Floodlight, POX and Maestro controllers.

I. Network DoS: Compromised forwarding device(s) direct traffic into a loop and magnify a flow until it completely fills out the available link bandwidth. As also reported in [21], we confirm that all four controllers were vulnerable to this attack and this completed in sub-second time intervals.

DETECTION: The attack involves a compromised forwarding device that either generates, misroutes or replays packets. These anomalies can be easily detected using the trajectory-based attack detection algorithms presented in §6.3. Compared to SPHINX, WedgeTail does not rely on any administrator defined policies for detection of a Network DoS attack.

II. Network-to-Host DoS: One or more forwarding devices send a large amount of traffic to the host network causing a DoS. This may bring down a host machine in extreme cases, and when dealing with mission critical systems, the impact would be catastrophic. Existing controllers do not have any detection mechanism against this attack.

DETECTION: Malicious forwarding device(s) may generate, replay or misroute packets towards a network host to cause a DoS attack. The aforementioned results into unexpected trajectories, which are detected by WedgeTail. However, unless there are administrator-defined policies for each host, SPHINX is unable to detect Network-to-Host DoS. Furthermore, the number of policies to be processed in real-time will be a factor of the total number of hosts and forwarding devices. The performance of SPHINX when processing such large number of policies is unknown. Moreover, even with such policies in place, the attack may go undetected as the downlink to host may not reach any suspicious threshold (note that in most cases this attack adds a negligible processing overhead to the compromised forwarding device(s) and may also have a negligible impact on the bandwidth).

III. TCAM Exhaustion: TCAM is a fast associative memory used to store flow rules. Malicious hosts may send arbitrary traffic and force the controller to install a large number of flow rules, thereby exhausting the switch's TCAM. As also discussed in §4, this may result in significant latency or packet drops. None of the controllers tested can detect nor prevent attacks such as TCAM exhaustion.

DETECTION: Attacks similar to III result in packet delay or drop. The latter will result in anomalies between expected and actual trajectories, which are detected by WedgeTail. Compared to WedgTail, SPHINX has a totally different approach in detecting such attacks. It checks for $FLOW\_MOD$ messages sent by the controller and detects a threat if the rate continues to be high over time. However, there may exist cases that the controller messages do not violate the administrator defined policies and still cause the switch to fail. For instance, the switch may be already experiencing a load higher, which may not be sought by an administrator when defining the policies. In such cases, the attack will not be detected by SPHINX.

IV. Forwarding device Blackhole: In this case, flow path ends abruptly, and the traffic cannot be routed to the destination. A forwarding device either drops or delays packet forwarding to launch this attack. We installed malicious rules on switches in networks, and none of the controllers had any mechanism to prevent nor detect them.

V. ARP Poisoning: Malicious network hosts can spoof physical hosts by forging ARP requests and fool the controller to install malicious flow rules to divert traffic. This may be used for eavesdropping or in other cases to mount IP slicing attacks and create network loops. We replicated the attack with the exact similar setup used in [21]. We also confirm that all of the tested controllers are vulnerable to it. Note that ARP poisoning corrupts the physical topological state. We discuss how WedgeTail detects attacks targeting the logical topological state in §10.

DETECTION: There are no network policies that a forged ARP request violates in a network. However, the actual path that a packet traveling from hosts to the controller takes is detected by WedgeTail. Hence, ARP requests with an anomalous trajectory (i.e. originated from hosts rather than forwarding devices) can be monitored and blocked before poisoning the network. SPHINX is also capable of detecting this attack either using its flow graph feature (which binds MAC-IP) or using administrator defined policies.

VI. Controller DoS: With OpenFlow, a packet that does not match any of the currently installed flow rules of a forwarding device is buffered, and an associated OFPT $PACKET\_IN$ message containing the data packet's header fields is forwarded to the controller. When a controller receives a large number of new packet flows within a short period, its buffer is filled up and has to forward complete packets to the controller. This causes heavy computational load on the controller, and it may bring it down altogether. We used Cbench [16] and flooded the controller with a high throughput of $PACKET\_IN$ messages to analyze the controllers' performance. Similar to [21], we report that all except Floodlight controller exhibited this attack. However, while Floodlight throttles the incoming OpenFlow messages from switches as a prevention mechanism, the connection of the switches with the controller is broken when a large number of switches attempt to connect with it.

DETECTION: A compromised forwarding devices may execute this attack by either replaying packets or generating packets destined to the controller. If WedgeTail detects an abnormal number of trajectories between a forwarding device and a controller it will detect a threat and can react as per the policies defined by its administrator. Note that WedgeTail may compute the threshold for the number of trajectories over time period $\Delta\tau$ by itself or, the administrator could custom define this. SPHINX detects a controller DoS by observing the flow-level metadata and computing the rate of PACKET_IN messages, which is compared with the administrator-defined policies. Compared to SPHINX, WedgeTail also has the advantage of computing the aggregated flow heading to the controller rather than each individual link.

9.3. Attack Implantations

WedgeTail successfully detected all of the attacks discussed in §9.2. However, to cover all of the malicious actions specified in §4 and perform extended performance analysis, we wrote scripts to implant 1500 synthetic malicious threats in our simulated networks. The resulting malicious forwarding devices maliciously processed: 1. All packets on all port, 2. All packets on a specific port, 3. A subset of packets on a all port, 4. A subset of packets on a specific port and 5. Packets destined to the control plane – resembling attacks against the control plane originating from the control plane. We regard detecting a forwarding device maliciously processing a subset of packets on a specific port as the most challenging case.

Malicious Actions: We used custom scripts to randomly introduce synthetic malicious forwarding devices in our networks. The resulting forwarding devices maliciously replayed packets (40% of all attacks), dropped packets (30%), misrouted packets (5%), generated packets (10%), and delayed packets (15%). A packet replay may be used in a range of threats (e.g. surveillance, DDoS, etc.) and is less likely to be detected compared to packet drops – i.e. traffic not reaching the destination is presumably much more noticeable. Hence, this distribution of attacks is deemed to be reasonable.

Compound Attack: Refers to the case where more than one malicious forwarding device are involved in an attack. For instance, a surveillance attack may involve more than one malicious forwarding device (see Figure 1). Compound attacks are challenging for solutions such as SPHINX to detect as compromised forwarding devices may intelligently install custom rules and avoid reporting to the controller thus aiming to conceal their maliciousness. This is less of an issue for WedgeTail's detection engine as any custom rule not matching those set by the control plane will eventually result in deviation of actual trajectories from expected ones.

In our simulations a total of 326 attacks involved more than one malicious forwarding devices (i.e. compound attacks). Specifically, 35% of these involved four malicious forwarding devices, 25% six forwarding device and 40% nine forwarding devices. In real-world scenarios, an attacker who has taken over nine forwarding devices of a network is relatively a strong and resourceful adversary. In AARNet Setup this means that the 75% of forwarding devices are compromised. Evidently, the aforementioned scenario is not supported by SPHINX, as authors in [21] assume the majority of forwarding devices to be non-malicious for their solution to work.

9.4. Accuracy & Detection Time

We measured WedgeTail's detection accuracy respect to the following three criteria:

A. Successful detection rate against attacks implanted in our simulated networks. We measured WedgeTail's detection accuracy in detecting attack scenarios discussed in §9.2 and the 1500 synthetic attacks implanted over our simulated networks. We report that all of the attacks have been successfully detected by WedgeTail.

B. Successful detection rate under network congestion leading to packet loss and delay. We added random congestion to the network leading to pack loss and delays at different nodes. We report that the packet delay and drop estimation mechanism employed (see §6.5) has minimized the impact of such in WedgeTail's accuracy. In fact, the performance was quite satisfactory with a relatively high detection accuracy (see Table 3).

C. Successful application of pre-defined policies against malicious forwarding devices. We report whenever a threat was detected and matching policies were specified, WedgeTail successfully applied them.

In A, the distribution of attacks over the networks was as following: 150 were over AARNet Setup, 750 over Zib54 Setup and 600 over Sprint Setup. We illustrate the detection time of 50 attacks separately over network AARNet Setup, B and C in Figure 5, 7 and 6, respectively. The average detection time over AARNet Setup is approximately 37 seconds with a standard deviation of 8 seconds. For Zib54 Setup, the average detection time is approximately 560 seconds with a standard deviation of 56 seconds. For Sprint Setup, the average detection time is approximately 4100 seconds with a standard 620 seconds. Moreover, the average detection times were not affected in the presence of Compound Attacks (see §9.3). The latter is expected since 1) the detection algorithm entails analyzing each and every forwarding device and 2) the response engine is not triggered until the end of full scan. Figure 9 shows the average detection time with respect to complexity of attacks present in Sprint setup evaluations. The x-axis values range between 0 (less complicated) and 1 (most complicated) to show the complexity of attacks. The most complicated attacks included multiple compromised forwarding devices maliciously processing a subset of packets on a specific port (see §9.3). As illustrated in Figure 9, WedgeTail's average detection time does not exceed 75 minutes even in the most complicated scenarios.

The aforementioned performance metrics show that WedgeTail's detection time scales well as the network size increases. Simply put, for an administrator of a medium-large size network being able to detect and locate malicious forwarding device after about half an hour without defining any policies or manual investigation is quite satisfactory.

We also report that despite the simulation size growth in terms of subnet, rules and trajectories the detection times have substantially reduced compared to WedgeTail 1.0. This is mainly

DD	A	FP	FN
3 minutes	98.83	3	0.76
5 minutes	99.17	3	0.69
10 minutes	99.38	8	0.48

Table 3: Overall detection results of attacks in the presence of packet drops due to congestion. In the table, DD: Detection Delay, A: Accuracy, FP: False Positive, FN: False Negative.

associated to: 1) improved scanning mechanism, 2) replacing HSA with NetPlumber and 3) the improvements introduced for attack detection algorithm (see §6.6).

For B, we added random congestion over simulated networks, which resulted in packet drops at various points in the simulated network. The dropped rate varied as 0, 0.005, 0.0075, 0.01, 0.015 and 0.02 of the 1K TCP flows sent over the simulated networks. Table 3 shows the overall detection results after detection delays of 3, 5 and 10 minutes – WedgeTail attack detection is started after the detection delay time. Note that we added multiple bottlenecks throughout the networks and The results prove that packet loss due to congestion is not a prohibitive factor for our system.

9.5. Performance Analysis

In this section, we report on some of the main performance metrics of WedgeTail. Thereafter, we compare WedgeTail's performance with related work and WedgeTail 1.0.

1. Scanning Mechanism: It is used to inspect the network and prioritize forwarding devices for inspection. As mentioned in §6.2, WedgeTail can be customized to prioritizes inspection within different user-defined time periods such as the last week, month and year. The algorithm used has two main steps that affect its efficiency index creation and TPMFP querying time. Index creation is only done once on the trajectory database being analyzed. The latter however is an online process executed on each request. In Sprint setup, with more than 1 million trajectories the index creation took a maximum of 20 minutes. Once the index are created, the query time is dependent on two steps: footmark graph construction and the time needed for searching most frequent path (MFP) in the graph. This process takes at most half a second for our largest dataset and few seconds in smaller setups. Therefore, the customization supported by WedgeTail and performance metrics are much improved compared to [53]. Figure 8 shows the processing time growth in respect to the number of trajectories.

2. Network Replica: We calculated the replication delays after 500 instances of updates in the original network, and we observed an upper bound of approx. 15 seconds. To the best of our knowledge, this is the first system to maintain a virtual network replica of an SDN data plane and might be an inspiring idea for future work.

3. Response Policy Matching: As shown in Figure 10, we observe that the average policy matching time as we increase WedgeTail's administrator defined policies from 0 to 1000 is approximately 120 milliseconds. Note that unlike SPHINX, WedgeTail's policies are used by its response engine only.

5. User Perceived Latencies: WedgeTail is not a real-time system, and it has no implication for the network users when detecting threats. Comparatively, however, SPHINX adds overhead to the network and causes delays. Given the various advantages of WedgeTail compared to SPHINX in detection and prevention, we consider this a bonus feature for our system.

Comparison with Related Work: We discuss the reasons as to why WedgeTail is non-comparable to network troubleshooting solutions in §10. However, to put WedgeTail's performance into perspective we report on the performance metrics of Anteater [42], which takes a snapshot of forwarding tables and analyze them for errors, and NetPlumber [29] that extends HSA into a real-time verification solution. Anteater has been tested on a 178 router topology and takes more than 5 minutes to just check for loops. NetPlumber may take up to 10 minutes to verify network correctness after a given rule change. Comparatively, WedgeTail investigates for every instance of malicious action and does much more than just evaluating rule sets (i.e. creating scanning regions, tracking packets as they traverse the network, maintaining a network replica for expected trajectory, etc.) with a reasonably added overhead.

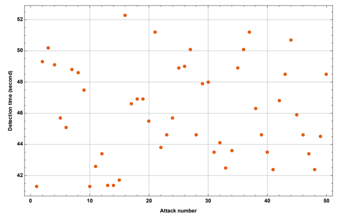


Figure 5: Attack detection in AARNet Setup (50 attacks).

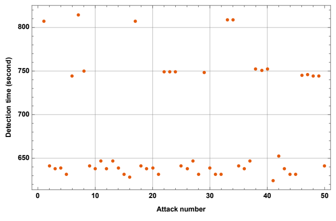


Figure 6: Attack detection in Zib54 Setup (50 attacks).

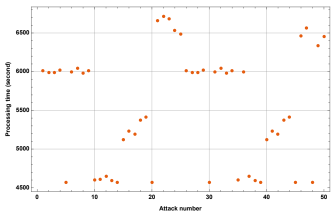


Figure 7: Attack detection in Sprint Setup (50 attacks).

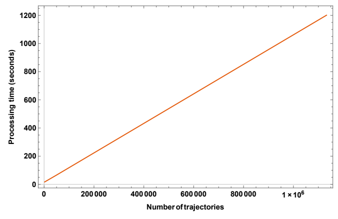


Figure 8: Scanning time in respect to the number of trajectories.

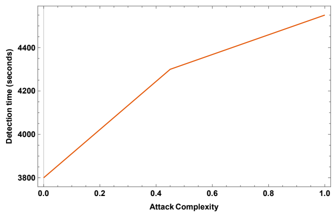


Figure 9: Detection time with respect to complexity of the attack (Sprint setup).

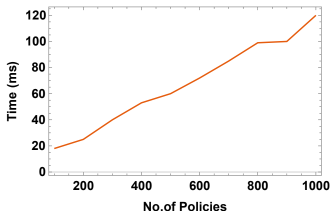


Figure 10: Average policy matching times with increasing policies.

10. The Good, the Bad and the Ugly

As mentioned in §1, the attack surface against forwarding devices has expanded over the last few years. However, today's routing protocols and network troubleshooting tools continue to assume the underlying hardware is trusted. Hence, networks require solutions to automatically detect malicious forwarding devices and protect the network from them irrespective of the cause and independent of underlying software and hardware.

Even with the latest network troubleshooting proposals, the main focus is on addressing issues such as configuration conflicts, routing loops, black holes and detection of policy inconsistencies (e.g. [31, 29, 47]). However, even with a correct configuration, the forwarding devices may fail in execution due to bugs in switch software, conflicts and limited memory space. Detecting forwarding devices not processing packets as inspected is challenging since common verification tools such as ping or traceroute 1) require extensive engineering over large networks, and 2) fail to detect forwarding devices cloaking their maliciousness.

10.1. Pre-WedgeTail

Related work including [21, 24, 17] 1) rely mainly on the administrator-defined policies for detection, 2) assume weak adversarial settings, and 3) fail to detect certain types of attacks (see §4 and §9.2). Moreover, they do not discuss the localization of malicious forwarding devices, impose overhead on network performance, cannot distinguish between malicious actions such as packet drop or delay and do not prioritize the inspection of forwarding devices.

10.2. Limitations and Post-WedgeTail

We evaluated WedgeTail over various network setups, configurations, and sizes equipped with different SDN controllers to prove its practicality under simulated environments. Specifically, WedgeTail's high accuracy and performance over Sprint Setup with a large number of forwarding devices, rules, and trajectories forms a solid ground motivating further development and evaluation of our proposed solution. Furthermore, we remind that WedgeTail's core detection and response techniques such as trajectory-creation, scanning methodology and inspection algorithms are platform independent and network dynamics do not alter these. Therefore, our next step is to deploy our solution over a real-world network setup focusing on scalability.

We also admit that we would need exploring WedgeTail's accuracy under more attack scenarios and use-cases (e.g. virtualization, VM migrations, and etc.). However, given our current evaluations results we do not expect any major hindrance for our steps forward.

Finally, WedgeTail's compatibility with distributed SDN controllers such as ONOS requires further investigation – although we regard such platforms to be an enabler rather than a barrier. We aim to address these limitations in the near future.

11. Conclusion

Information is the new gold, it is the new oil. It is worth trillions of dollars and whoever controls it has the power to control wealth and power. Nowadays, Internet is much more than a source of information, it is the building block of today's democracy. Freedom is embedded in this technology with tools such as blogs and social networks and we have to protect this technology. In the era of cyber-war and cyber-terrorism, attackers are targeting the very core of today's network infrastructure including the network routers. Software Defined Networks (SDN) is regarded as the networks of the future and it must be secured. The SDN control plane security has been an ongoing topic of research. However, malicious forwarding devices could potentially be a more worrying threat as these are the actual enforcement point of decisions made at the control plane. Accordingly, SPHINX [21] was the first attempt in the literature to detect a broad class of attacks in SDNs with a threat model not requiring trusted switches or hosts. With the same set of goals, we proposed an alternative solution, which we call WedgeTail. Our solution is designed against stronger adversarial settings and outperforms prior solutions in various aspects including accuracy, performance, and autonomy.

References

References

[1] Mausezahn. `http://www.perihel.at/sec/mz/`.
[2] Open Networking Foundation (ONF). `https://www.opennetworking.org/`.

[3] Chinese hackers who breached Google gained access to sensitive data, U.S. officials say. https://www.washingtonpost.com/world/national-security/chinese-hackers-who-breached-google-gained-access-to-sensitive-data-us-officials-say/2013/05/20/51330428-be34-11e2-89c9-3be8095fe767_story.html?utm_term=.91c0c5b17299, 2013.

[4] Huawei HG8245 backdoor and remote access. http://websec.ca/advisories/view/Huawei-web-backdoor-and-remote-access, 2013.

[5] Netis Routers Leave Wide Open Backdoor. http://blog.trendmicro.com/trendlabs-security-intelligence/netis-routers-leave-wide-open-backdoor, 2014.

[6] NIST: CVE-2014-9295 Detail. https://nvd.nist.gov/vuln/detail/CVE-2014-9295, 2014.

[7] Snowden: The NSA planted backdoors in Cisco products, Infoworld. http://infoworld.com/article/2608141/internet-privacy/snowden--the-nsa-planted-backdoors-in-cisco-products.html, 2014.

[8] NSA Preps America for Future Battle, Spiegel. http://www.spiegel.de/international/world/new-snowden-docs-indicate-scope-of-nsa-preparations-for-cyber-battle-a-1013409.html, 2015.

[9] SYNful Knock - A Cisco router implant - Part I. https://fireeye.com/blog/threat-research/2015/09/synful_knock_-_acis.html, 2015.

[10] Cisco IOS and IOS XE Software Cluster Management Protocol Remote Code Execution Vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170317-cmp, 2017.

[11] NIST: CVE-2014-9295 Detail. https://nvd.nist.gov/vuln/detail/CVE-2014-9295, 2017.

[12] Vault 7: CIA Hacking Tools Revealed. https://wikileaks.org/ciav7p1, 2017.

[13] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha. A survey of securing networks using software defined networking. *IEEE transactions on reliability*, 64(3):1086–1097, 2015.

[14] F. Assolini. The tale of one thousand and one dsl modems, kaspersky lab, 2012.

[15] K. Benton, L. J. Camp, and C. Small. Openflow vulnerability assessment. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 151–152. ACM, 2013.

[16] Cbench. *https://goo.gl/10TLJk*.

[17] T.-W. Chao, Y.-M. Ke, B.-H. Chen, J.-L. Chen, C. J. Hsieh, S.-C. Lee, and H.-C. Hsiao. Securing data planes in software-defined networks. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 465–470. IEEE, 2016.

[18] D. Chasaki and T. Wolf. Attacks and defenses in the data plane of networks. *IEEE Transactions on dependable and secure computing*, 9(6):798–810, 2012.

[19] CRATE datasets. *ftp://download.iwlab.foi.se/dataset*.

[20] Data Set for IMC 2010 Data Center Measurement. *http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html*.

[21] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS*, 2015.

[22] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 271–282. ACM, 2000.

[23] A. Feldmann, P. Heyder, M. Kreutzer, S. Schmid, J.-P. Seifert, H. Shulman, K. Thimmaraju, M. Waidner, and J. Sieberg. Netco: Reliable routing with unreliable routers. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 128–135. IEEE, 2016.

[24] R. Ghannam and A. Chung. Handling malicious switches in software defined networks. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, pages 1245–1248. IEEE, 2016.

[25] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 71–85, 2014.

[26] P. Hunter. Pakistan youtube block exposes fundamental internet security weakness: Concern that pakistani action affected youtube access elsewhere in world. *Computer Fraud & Security*, 2008(4):10–11, 2008.

[27] W. Jiang, J. Zhu, J. Xu, Z. Li, P. Zhao, and L. Zhao. A feature based method for trajectory dataset segmentation and profiling. *World Wide Web*, 20(1):5–22, 2017.

[28] A. Kamisiński and C. Fung. Flowmon: Detecting malicious switches in software-defined networks. In *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, pages 39–45. ACM, 2015.

[29] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, 2013.

[30] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, 2012.

[31] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, 2013.

[32] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig. Lightweight source authentication and path validation. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 271–282. ACM, 2014.

[33] R. Klöti, V. Kotronis, and P. Smith. Openflow: A security analysis. In *21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2013.

[34] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[35] D. Kreutz, F. Ramos, and P. Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 55–60. ACM, 2013.

[36] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[37] LBNL/ICSI Enterprise Tracing Project. *http://www.icir.org/enterprise-tracing/*.

[38] S. Lee, T. Wong, and H. S. Kim. Secure split assignment trajectory sampling: A malicious router detection system. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 333–342. IEEE, 2006.

[39] K.-C. Li, H. Jiang, L. T. Yang, and A. Cuzzocrea. *Big data: Algorithms, analytics, and applications*. Chapman and Hall/CRC, 2015.

[40] F. Lindner. Cisco ios router exploitation. *Black Hat USA*, 2009.

[41] W. Luo, H. Tan, L. Chen, and L. M. Ni. Finding time period-based most frequent path in big trajectory data. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 713–724. ACM, 2013.

[42] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 290–301. ACM, 2011.

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

[44] A. T. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Fatih: Detecting and isolating malicious routers. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 538–547. IEEE, 2005.

[45] A. T. Mizrak, S. Savage, and K. Marzullo. Detecting malicious packet losses. *IEEE Transactions on Parallel and distributed systems*, 20(2):191–206, 2009.

[46] Open Networking Foundation (ONF). Sdn architecture, onf tr-502. *opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf*.

[47] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski. Sndlib 1.0–survivable network design library. *Networks*, 55(3):276–286, 2010.

[48] N. Pelekis, I. Kopanakis, C. Panagiotakis, and Y. Theodoridis. Unsupervised trajectory sampling. In *Machine learning and knowledge discovery in databases*, pages 17–33. Springer, 2010.

[49] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review*, 44(4):407–418, 2015.

[50] Route Views. *http://www.routeviews.org*.

[51] D. Sacharidis, K. Patroumpas, M. Terrovitis, V. Kantere, M. Potamias, K. Mouratidis, and T. Sellis. On-line discovery of hot motion paths. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 392–403. ACM, 2008.

[52] S. Scott-Hayward, S. Natarajan, and S. Sezer. A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, 18(1):623–654, 2015.

[53] A. Shaghaghi, M. A. Kaafar, and S. Jha. Wedgetail: An intrusion prevention system for the data plane of software defined networks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 849–861, New York, NY, USA, 2017. ACM.

[54] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Transactions on networking*, 12(1):2–16, 2004.

[55] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 228–237. IEEE, 2014.

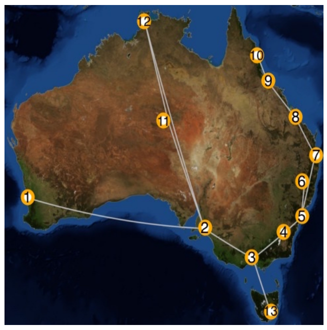
[56] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 87–99, 2014.

[57] X. Zhang, C. Lan, and A. Perrig. Secure and scalable fault localization under dynamic traffic patterns. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 317–331. IEEE, 2012.

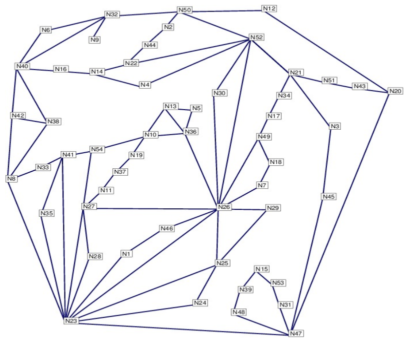
[58] X. Zhang, Z. Zhou, H.-C. Hsiao, T. H.-J. Kim, A. Perrig, and P. Tague. Shortmac: Efficient data-plane fault localization. In *NDSS*, 2012.

[59] Y. J. Zhu and L. Jacob. On making tcp robust against spurious retransmissions. *Computer communications*, 28(1):25–36, 2005.

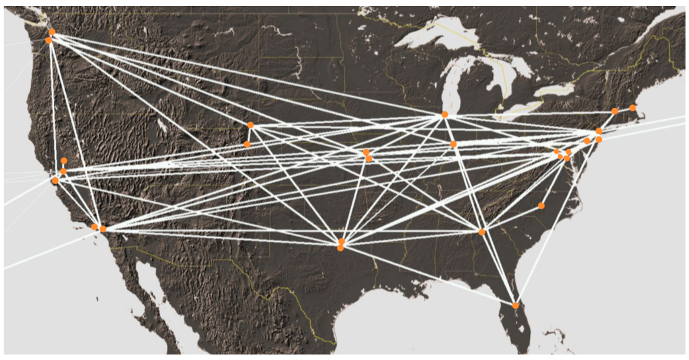
Appendix A. Network Topologies used in Simulations



a



b



c

Figure A.11: Network Topologies used in WedgeTail Evaluations.