WedgeTail: 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, an Intrusion Prevention System (IPS) designed to secure the SDN data plane. WedgeTail regards forwarding devices as points within a geometric space and stores the path packets take when traversing the network as trajectories. To be efficient, it prioritizes forwarding devices before inspection using an unsupervised trajectory-based sampling mechanism. For each of the forwarding device, WedgeTail computes the expected and actual trajectories of packets and 'hunts' for any forwarding device not processing packets as expected. Compared to related work, WedgeTail is also capable of distinguishing between malicious actions such as packet drop and generation. Moreover, WedgeTail employs a radically different methodology that enables detecting threats autonomously. In fact, it has no reliance on pre-defined rules by an administrator and may be easily imported to protect SDN networks with different setups, forwarding devices, and controllers. We have evaluated WedgeTail in simulated environments, and it has been capable of detecting and responding to all implanted malicious forwarding devices within a reasonable time-frame. We report on the design, implementation, and evaluation of WedgeTail in this manuscript.

Keywords

Software Defined Networks; SDN Security; Data Plane Security; Intrusion Prevention System

1. INTRODUCTION

An attacker may compromise a network forwarding device by exploiting its software or hardware vulnerabilities. Compromised forwarding devices may be then used to drop or slow down, clone or deviate, inject or forge network traffic to launch attacks targeting the network operator and its users. As discussed in [22, 24], compromised forwarding devices may even grant an attacker the capability to wrest control of an entire Software Defined Network (SDN). This paper looks 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 have not been the subject of many studies in SDN security research – see [3, 25, 38] for comprehensive surveys of SDN security. In fact, even with the latest proposals, there seems to be an oversight regarding the malicious forwarding devices that may exist in SDN data plane [13]. In general, the development of SDN security applications and controllers and real-time verification of network constraints, separately, have been the primary focus of SDN security literature (see [3, 25] for surveys). However, no combination of these provides effective protection against compromised forwarding devices [8, 11, 13].

Recently, a few proposals specifically look into the threats associated with malicious forwarding devices. However, these either suffer from a simplistic threat model (e.g. [17]) or substantial processing overhead imposed to the network (e.g. [21, 42, 43]). For example, cryptographic solutions such as [21] 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, increased packet size, and etc.

SPHINX [11] is one of the solutions designed for securing the SDN data plane that does not assume forwarding devices are to be trusted. SPHINX detects and mitigates security attacks launched by malicious switches by abstracting the network operations with incremental flow graphs. It detects attacks as per the policies defined by the administrator and responds accordingly. SPHINX 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. Therefore, SPHINX does not assume malicious forwarding device could behave arbitrarily and is not capable of distinguishing between malicious actions (e.g. packet drop and fabrication), and it cannot detect when a malicious forwarding device is delaying packets. Second, the detection mechanism mainly relies on the policies defined by an administrator to detect attacks. In fact, the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3052973.3053039

flow-graph component does not validate forwarding device actions against the controller policies but only compared to their behavior over time – hence, radical network configuration changes will lead to false positives. Moreover, the flow-graph feature requires that the majority of forwarding devices be trustworthy. Indeed, an alternative more robust solution will have to be independent of this assumption. Thirdly, SPHINX does not prioritize its inspection of forwarding devices. Arguably, an efficient solution should prioritize this task to improve detection performance.

Here, we introduce WedgeTail, a controller-agnostic Intrusion Prevention System (IPS) designed to 'hunt' for forwarding devices failing to process packets as expected. WedgeTail regards packets as 'random walkers' [30] in the network and analyzes packet movements as trajectories in a geometric space. By analyzing the expected and actual trajectories of packets, our proposed solution is capable of automatically localizing malicious forwarding device and identifying the exact malicious behavior (e.g. packet drop, fabrication). WedgeTail response to threats can be programmed using administrator-defined policies. For example, an instant isolation policy may be customized such that initially, the potentially malicious device is instructed to reset all the flow rules and then, evaluated at various intervals by re-iterating the same packet(s) raising suspicion.

In order to make the scanning more efficient and increase the probability of finding malicious devices earlier, WedgeTail begins by prioritizing forwarding for inspection. We adopt Unsupervised Trajectory Sampling [35] to cluster forwarding devices into scanning groups of varying priority depending on the cumulative frequency of occurrence in packet paths traversing the network. To retrieve the expected trajectories, WedgeTail intercepts the relevant OpenFlow messages exchanged between the control and data plane and maintains a virtual replica of the network. This virtual replica is processed by its integrated Header Space Analysis (HSA) [19] component to calculate the expected packet trajectories. The actual packet trajectories are, however, computed by tracking a custom hash of the packet header. Alternatively, if NetSight [15] is deployed, WedgeTail queries for packet history to retrieve the packet trajectory. We briefly review [15] and [19] in §2.

The contributions of this work can be summarised as follows:

(a) We define an advanced threat model for the security of SDN data plane that has not been considered up to now (§3). In §4, we first discuss the main factors that exacerbate the protection of SDN networks against malicious forwarding devices. Thereafter, the requirements for an effective solution and the key insights behind our proposed solution is presented.

(b) In §5, we present WedgeTail's target identification mechanism, where we detail how to retrieve the packet trajectories and analyze them to create scanning regions.

(c) In §6, we present our proposed attack detection algorithms and localization logic. We also discuss how WedgeTail distinguishes between different malicious packet processing actions (e.g. packet replay and drop). WedgeTail's response engine and its capabilities are discussed in §7.

(d) We discuss WedgeTail's implementation in §8. Thereafter, in §9, we evaluate WedgeTail's performance and accuracy over three different simulated networks. We conclude the paper by comparing our solution with related work and outlining the future work (§10).

2. BACKGROUND

2.1 Header Space Analysis (HSA)

Header Space Analysis (HSA) [19] is a method 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 the path a packet traversing the network on a certain port will take. We have included an example usage of HSA and how it serves for predicting packet trajectories in §4.

2.2 NetSight

NetSight [15] 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 §6 we show how WedgeTail may inter-operate with NetSight to retrieve the actual packet trajectories.

3. 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, [11, 18–20, 28], 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 [11], the closest work to ours. 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 cannot 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 even generate (includes both modify and fabricate) packets, in random or selective manner 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 (§11.

We regard a forwarding device as 'malicious' when both of the following properties are met: A) The device is not handling the network packets according to the rules specified by the control plane. 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 a faulty, forwarding device too. 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 could also 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 itself and the defined policies are trustworthy and securely transferred to the data plane (e.g. using TLS protocol [6]). There is an increasing body of literature aiming to achieve this, see [3, 38] for surveys. In other words, with SDN, the policy definition and enforcement points are separated in networks [25] and here, we exclusively focus on the the Policy Enforcement Point (PEP). Hence, preventing incidents such as [16] caused by erroneous administrator defined policies is out of scope.

2. Packet reordering and time behaviour [31] are well-studied and proposed solutions are complementary to WedgeTail. This is also true regarding protocol-level attacks including TCP/IP and OSPF that can be addressed using existing solutions. In fact, WedgeTail is designed to detect forwarding devices failing to execute their main function and not to protect them from being compromised. In other words, the prevention refers to the automated triggering of pre-defined policies against identified threats.

3. The forwarding devices may lie about anything except their own identity – similar assumption is also made in [11].

4. WEDGETAIL

As mentioned in §1, securing SDN networks against malicious forwarding devices is challenging. In fact, similar to [11], we also argue that the problem of protecting networks and their host against malicious forwarding devices is exacerbated in SDN context. We believe this due to five main reasons – the first three factors are extracted from [11] with some minor amendments and additions.

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. [11] postulates that for a comprehensive defense against traditional attacks either a fundamental redesign of OpenFlow [29] protocol would be required, or we would need to patch the controller per each attack.

The second factor is the unverified and complete reliance of control plane on forwarding devices. An SDN controller relies on $PACKET_IN$ messages for its view of the network, yet this is not securely authenticated nor verified. A malicious forwarding device may send forged spoofed messages to subvert the controller view of the network – even with having TLS authentication in place. The same vulnerability enables a compromised forwarding device the capability to overload the controller with requests causing a Denial of Service (DoS) attack.

Third, securing programmable soft-switches such as Open vSwitches is more challenging compared to hardware equivalents. The former run atop of end host servers and are more susceptible to attacks compared with the hardware switches, which is harder for an attacker to physically access.

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

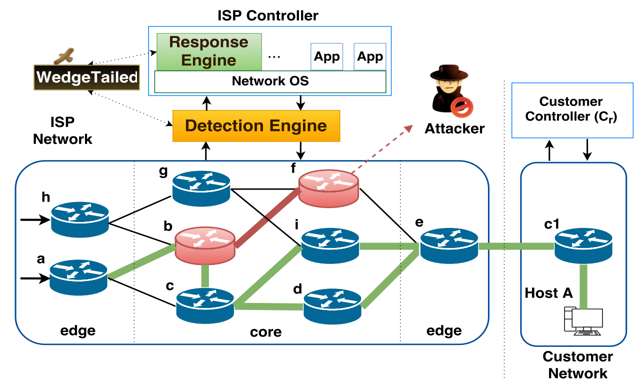


Figure 1: Forwarding devices in ISP Network. The red devices are misbehaving. The green links are expected paths.

Fifth, motivated by performance advantages such as lower latency response to network events and better protocol standardization such as for encryption, MAC learning and codec control message (CCM) exchanges, proposals have been made to delegate more control to the SDN data plane [33]. The increased authority improves the network's fault tolerance with the continuation of basic network operations under failing controllers. However, this increases the vulnerability of the network to traditional attacks and expands the range of attacks a compromised device may launch against the network.

Considering the aforementioned factors and the limitations of existing work such as SPHINX (see § 1), we posit the 'must-have' features of an effective solution against malicious forwarding devices to include:

1) Minimum reliance on pre-defined rules/policies for detection: to be able to detect unknown attacks exploiting either hardware or software vulnerabilities of forwarding devices,
2) Ability to systematically and autonomously prioritize forwarding devices for inspection: to improve detection performance and success rate,
3) Capability to distinguish malicious forwarding actions and localize maliciousness: to avoid executing conflicting and redundant policies when responding to threats,
4) Programmability for responding to threats: to be able to customize response when detecting threats, and
5) Causing minimal disruption to the network performance when detecting and responding to threats: so the proposed solution is practical for real-world network deployment.
The rest of this paper describes how we address each of the aforementioned requirements in our proposed solution, WedgeTail.

4.1 Overview

As shown in Figure 1, WedgeTail is composed of two main parts namely, Detection Engine (§6) and Response Engine (§7). The former listens to OpenFlow messages exchanged between the control and data plane and by doing so maintains a virtual replica of the network, which is used to compute the expected packet trajectories. The Response Engine, however, is placed as an application on top of the controller and submits policies to the network operating system, which makes the final decision on how, and whether, to apply them.

Now, assume an ISP network with AARNet Setup traffic flow as shown in Figure 1. Using its integrated HSA component, WedgeTail retrieves 10010x10 ∪ 10011x10 as the

header space of packets that can be sent from Forwarding Device (FD) a to c1, or $FD(a) \rightarrow FD(c1)$, on $Port_i$. Composing the history attributes of the propagation graph it learns that the Expected Packet Trajectories between these two nodes is as follows:
$FD(a) \rightarrow FD(b) \rightarrow FD(c) \rightarrow \{FD(d)\ OR\ FD(i)\} \rightarrow FD(e) \rightarrow FD(c1)$ – shown in green colour in Figure 1.

The main intuition behind WedgeTail is that whenever the *Actual Packet Trajectories* are not a subset of the *Expected Packet Trajectories*, one or more of the forwarding devices in the packet path may be malicious – recall that in §3 'malicious' was extended to cover faulty and misconfigured too. 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 whole network on a specific port – out of the designated target ports. The detection engine entails Find-Target-Forwarding-Devices() and Scan-for-Attacks(), which provide input to the main protection engine function, Isolate-Forwarding-Device().

Algorithm 1 WedgeTail Detection and Response

```
Response Policy RP
Select Port_i ∈ {Port}
Find-Target-Forwarding-Devices (Port_i)
Select FD(i) ∈ {Target Forwarding Devices}
for all Port_i ∈ {Port} do
   Scan-for-Attacks (FD(i))
   if FD(i) is 'Malicious' then
      Isolate-Forwarding-Device (RP, FD(i))
   end if
end for
```

5. TARGET IDENTIFICATION

5.1 Trajectory Creation

Definition of Trajectory: We first define the notion of trajectory, denoted as TR hereon, in the context of our work. A packet trajectory is the route a uniquely identifiable packet takes while traversing a network from one forwarding device to another. We consider different paths for the same packet as distinctive trajectories. In other words, a packet may be routed through various paths in respect to network configurations and condition on each iteration. For instance, as shown in Figure 1, a packet traversing through green line from $FD(a)$ to $FD(e)$ may be routed through $FD(i)$ or $FD(d)$ depending on the QoS requirements. However, multiple repetitions of the same path for the same packet is only regarded as one trajectory.

Retrieving the Actual Packet Trajectories: We propose two alternative 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 in a network, a convenient approach would be to query for each packet header route and create the trajectories. This may be achieved by integrating a simple module for WedgeTail, which uses existing API provided by NetSight. Our preferred method to retrieves the actual trajectories is through NetSight.

An alternative approach would be for WedgeTail to run a deterministic hash function over the packet header and use

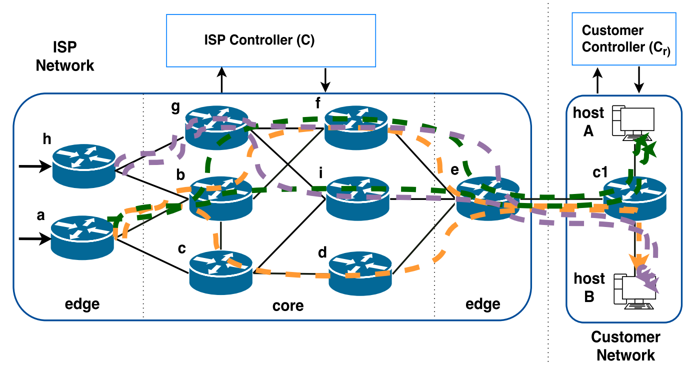


Figure 2: Forwarding devices in ISP network. The dotted lines represent traffic paths.

this hash to track packet as it traverses the network (i.e. generating 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 [12]. We then pick packet header values such as source address, destination address from IP header and source port and destination port in TCP header. All 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. Evidently, in this case, the overhead to collect trajectory samples is small since the traffic that has to be collected from nodes only consists of such labels (plus some auxiliary information) [12].

An issue to consider is that in the unlikely case that collisions were to occur, WedgeTail's performance will not be affected. This is because such collisions will break the order of forwarding devices when retrieving trajectories and will result in invalid trajectories. Moreover, we envision the hashing-based solution to be used as an alternative where NetSight is not available and at most within small networks, where collisions are much less likely to occur.

5.2 Scanning Zones

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 identifies the most commonly involved forwarding devices by looking at the denser regions. For instance, looking at Figure 2 and the drawn trajectories, 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 a huge number of trajectories. Therefore, WedgeTail reduces this large set into a representative sample that encapsulates the most commonly visited forwarding devices. Formally, let $TR[FD(i), FD(j)]$ denote the set of all the trajectories traversing between $FD(i)$ and $FD(j)$ for all packets on all ports. Accordingly, define $\{TR(N)\}$ as the set of all trajectories in network N, or $\{TR(N)\} = \{TR[FD(i), FD(j)] \,|\, \forall (FD(i), FD(j)) \in N\}$. Denote $\{TR(N)\}'$ as a subset of $\{TR(N)\}$, which if inspected by WedgeTail without loss of generality results in detecting compromised forwarding devices. Indeed, such sampling is challenging due to the complexity of packet routing (e.g. lack of ordering, lack of compact representation). To

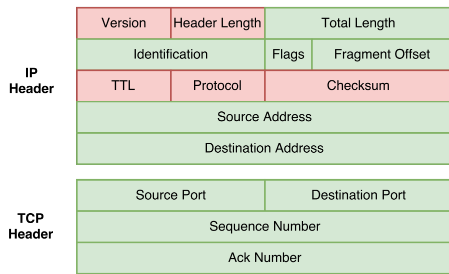


Figure 3: Packet header fields used for labeling.

automatically compute this, in an optimised and unsupervised way, WedgeTail adapts the Unsupervised Trajectory Sampling technique proposed in [35].

The three-step Unsupervised Trajectory Sampling solution proposed in [35] may be summarized as follows. The first step involves adopting a symbolic representation of trajectories to model all of the collected trajectories in an approximate way as vectors. The symbolic representation is lossless in terms of mobility patterns and improves the speed of computation. Thereafter, on top of the representation, each trajectory is represented using a continuous function that implicitly describes the representativeness of each constituent part of it with respect to all of the collected trajectories. Symbolic Trajectory algorithm, or SyTra, is used to improve the initial representation of each trajectory by relaxing its vector representation. 'The idea is to adopt a merging algorithm that identifies the maximal time period wherein the mobility pattern of each trajectory is preserved, while in this augmented period it presents uniform behavior in terms of representativeness' [35]. In the third step, an automatic method for trajectory sampling, known as T-Sampling, based on the representativeness of the trajectories is used. T-Sampling takes into account not only the most (i.e., dense, frequent) but also the least representatives. This is an important aspect of this work, which makes it the best match for our requirements. In fact, alternative sampling techniques suffer from shortcomings that limit their application for WedgeTail. For example, [4,5] are explorative and supervised sampling techniques that assume a priori knowledge of the underlying trajectories. Alternatively, [14, 27] downsize the collection of trajectories and fail to select trajectories important for mobility patterns.

Once the most commonly visited forwarding devices are extracted from the network they are allocated the highest priority of inspection and the remaining forwarding devices are assigned a lower priority for inspection.

6. ATTACK DETECTION

The main attack detection algorithm (*Scan-for-Attacks*) is presented in Algorithm 2. The algorithm takes as input both a target forwarding device and a port and returns a malicious node detailing 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 §5.1. 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 $\not\subseteq$ E then $FD(i)$ is malicious. The comparison logic can be extended to differentiate between the four types of malicious actions (see §3) as follows:

Algorithm 2 Attack Detection Algorithm

```
Scan-for-Attacks(FD(i), PortP_i) {
Status S = Check-State-Change();
File F = Dataplane-Configrautions-Snapshot(S);
while Check-State-Change() == S do
  List L = F.ForwardingDevices() – FD(i)
  for all FD(j) ∈ 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 ≠ Expected then
      Identify-Attack(FD(i), Port(i));
    end if
  end for
end while
}
```

Here, without loss of generality we assume, there exists only one valid trajectory between two forwarding devices.

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, 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 and authentication attacks.

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 $A' \not\subseteq E$ then WedgeTail detects a packet misrouting attack.*

3. Packet Dropping: A compromised forwarding device that drops packets creates a black or gray 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. WedgeTail detects packet dropping as follows:

Detection: *If $A \not\subseteq E$ 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. WedgeTail detects packet generation as follows:

Detection: *If $A \not\subseteq E$ and $E - A = E$.*

5. Packet Delay: Occurs when a forwarding device delays traffic and increases jitter. A packet delay is a threat against time-sensitive traffic [13]. A delay of TCP stream also causes spurious timeouts and unnecessary re-transmission, which severely threatens the TCP throughput [44].

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 [36] or [40], where it is possible to achieve real-time congestion detection and measurement.

6.1 Malicious Localization

As mentioned a trajectory is regarded as a total ordered set. Once one of the malicious actions is 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.

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

$E(\bar{\tau})$: $FD(a) \to FD(b) \to FD(c) \to FD(d) \to FD(e)$ 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 $FD(d)$ (or, $A - E$,) 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 however, we retrace time hop by hop in A and compare with the relevant expected time.

6.2 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. [32] can 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 [32] is that we could implement it without any additional overhead or support from the network – [32] assumes the routers in the network cooperate and provide real-time data related to the queue lengths at their interfaces.

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
Validity	t (millisecond)

Table 1: Overview of the response engine policy syntax

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 Validity 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 validity is used to specify timing.

We now look at two examples. Let us revisit 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. 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, evaluate it over different controllers and to ease the implementation we implemented the detection engine as a proxy sitting in between the control and data plane – a similar architecture is also used in [11]. In fact, the detection engine requires advanced functions that, currently, is not consistently available across controllers. Currently, the response engine is programmed as an application for Floodlight controller. WedgeTail's current architecture is shown in Figure 4.

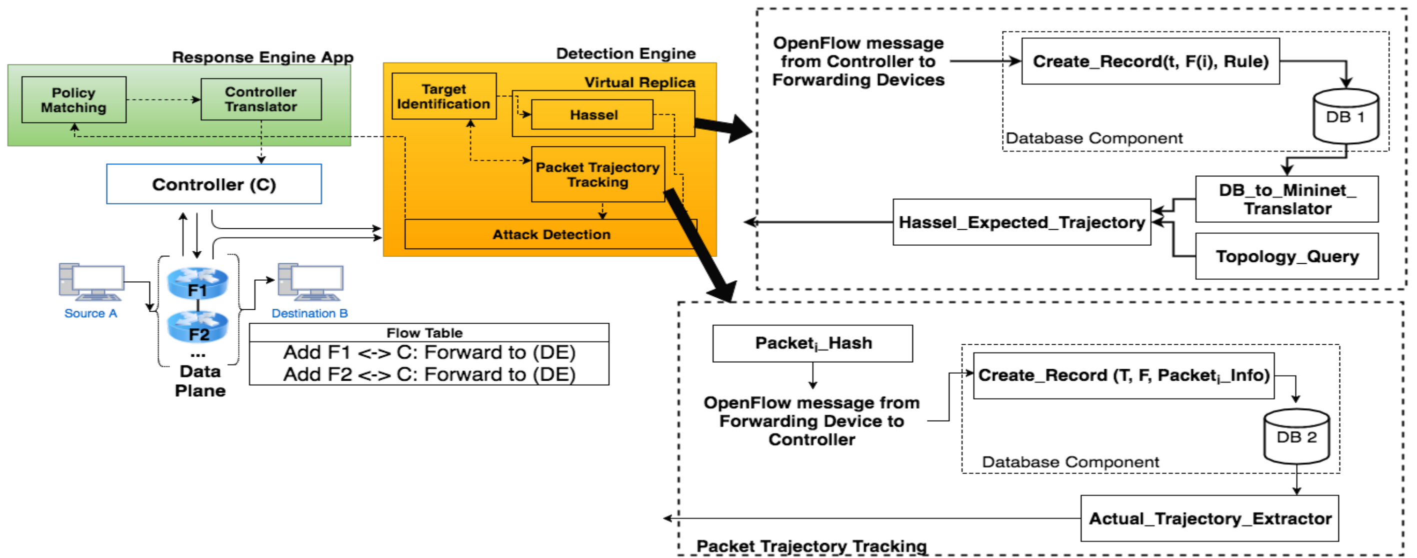


Figure 4: WedgeTail Architecture

We implemented our system, mainly, in Java using approximately 1500 lines of code. WedgeTail work starts by creating 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. This information is then used to create database records that list all the forwarding devices visited by packets along with some packet information and a timestamp attached to it. WedgeTail composes these records together and generates the actual packet trajectories using its Actual_Trajectory_Extractor module.

Once the scanning zones are generated, and the target forwarding devices identified, WedgeTailed requires having the expected trajectories of packets to initiate its inspection. Hence, WedgeTail 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 the Hassel_Expected_Trajectory module to compute the expected trajectories of packets[1].

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 WedgeTail in detecting these attacks. In order to further evaluate WedgeTail's detection engine, we then wrote scripts that synthetically implanted a total 500 attacks covering all of the malicious actions specified in §3 in our simulated networks.

[1]We acknowledge that the authors of [35] provided the source code of their program, which we used in our Target Identification module.

Number of	AARNet Setup	Zib54 Setup	Sprint Setup
FD	12	54	316
Subnet	40	800	48,966
Rules	391	21,387	15,649,486
Trajectory	403	38,654	638,271

Table 2: Overview of simulated networks

Here, we report on WedgeTail's accuracy and performance including its detection and prevention success, average detection time, user perceived latencies, overheads related to policy verification, etc. To further challenge WedgeTail's detection engine, we also introduced congestion to the simulated networks causing packet losses and measured the resulting false alarms. Moreover, we also compared our proposal with related works and argued how WedgeTail, in most cases, outperforms them both in detection and response. 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.

9.1 Experimental Setup

We simulated three different networks namely *AARNet Setup*, *Zib54 Setup* and *Sprint Setup*. AARNet Setup was used in our initial feasibility study and resembled a minimalistic backbone ISP network topology with only 12 forwarding devices. The forwarding rules in this network reached 390, and we generated benign traffic such that about 400 trajectories existed in the snapshots taken for inspection by WedgeTail. In Zib54 Setup, we extended our simulated network to more than 50 forwarding devices. Snapshots are taken for

analysis contained up to 21000 forwarding rules and 38000 trajectories over 800 subnets. A large network is presumed to have more forwarding devices as well as many more trajectories at any given instance. These motivated us to evaluate WedgeTail under Sprint Setup. The Sprint Setup contained 316 forwarding devices with more than 600000 trajectories over a network with more than 15 million rules and 48000 subnets.

Network Topologies: The network topologies for AARNet Setup, Zib54 and Sprint were extracted from The Internet Topology Zoo [23], SNDlib [34] and Rocketfuel [39], respectively. Figure 11.a and 11.b (in Appendix) 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) 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 were 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 [37] 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 relevant work such as [8], [41].

Traffic Generation: We used Mausezahn [1] and a custom script to add benign traffic to the networks. Similar to [11], our custom written script imported three real-world network traces from [9, 10, 26] to drive traffic into Mininet.

We hosted the simulated networks on a machine equipped with Intel Core i5, 2.66 GHz quad-core CPU and 16 GB of RAM. The SDN controller equipped with WedgeTail was hosted on a machine with Intel Core i7, 2.66 GHz quad-core CPU and 8 GB of RAM.

9.2 Attack Scenarios

We replicated all the attacks presented by the authors of [11] against networks using ODL, Floodlight, POX and Maestro as controllers. We also implemented further attacks including Network-to-Host DoS attack. All of these attacks use one or more of the capabilities defined in §3 and were successfully detected by WedgeTail. As examples, we briefly revise the main characteristics of six different attacks and discuss how WedgeTail successfully detects all of them. We also compare the advantages of WedgeTail with SPHINX when detecting these threats – note that the authors did not provide the source of their solution and therefore, we cannot provide a numerical performance comparison at this stage. We report on the performance metrics separately in §9.4 and §9.5.

I. Network DoS: In this case, compromised forwarding devices direct traffic into a loop and magnify a flow until it completely fills out the available link bandwidth. We report that all four controllers were vulnerable to this attack and this completed in sub-second time intervals.

DETECTION: This attack involves a compromised forwarding device that either generates, misroutes or replays packet(s). These anomalies can be easily detected using the trajectory-based attack detection algorithms presented in §6. Compared to SPHINX, WedgeTail does not rely on any administrator defined policies for detection of a Network DoS attack.

II. Network-to-Host DoS: Here, 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 result of the aforementioned actions is having unexpected trajectories in the network, which are automatically 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 into installing a large number of flow rules, thereby exhausting the switch's TCAM. As also discussed in 3, 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 results in packet delay or drop, which will result in anomalies between expected and actual trajectories and are detected by WedgeTail. SPHINX has a totally different approach for detecting such attacks. The latter checks for $FLOW_MOD$ messages sent by the controller and detects a threat if the rate continues to be high over time. While both approaches will lead to detecting the threat, with SPHINX, the controller messages may not violate the administrator defined policies and still cause the switch to fail (e.g. the switch may be already experiencing a load higher than usual that is not covered in the policy description). 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 creating network loops. We replicated the attack with the exact similar setup used in [11] and we also report 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 ac-

tual path that a packet traveling from hosts to the controller takes is visible to 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 detection 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 [7] and flooded the controller with a high throughput of $PACKET_IN$ messages to analyze the controllers' performance. Similar to [11], we report that all except Floodlight 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 there are an abnormal number of trajectories between a forwarding device and a controller in a snapshot taken from the network, then WedgeTail will detect a threat and can react as per the policies defined by its administrator – note that WedgeTail may compute the threshold 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

As mentioned, WedgeTail successfully detected all of the attacks implemented in §9.2. However, to cover all of the malicious actions specified in §3 and perform extended performance analysis, we wrote scripts to implant 500 synthetic malicious threats in our simulated networks. The resulting malicious forwarding devices maliciously processed: **1.** All packets on all ports in approx. 30% of all attacks, **2.** A subset of packets on a specific port in approx. 19% of all attacks, **3.** A subset of packets on a specific port in approx. 19% of all attacks, **4.** Packets pertaining to a specific port in approx. 25% of all attacks, **5.** A subset of packets pertaining to a specific port in approx. 15% of all attacks, **6.** Packets destined to the control plane in approx. 11% of all attacks.

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 Attacks: We define compound attacks as those involving more than one malicious forwarding device. For example, a surveillance attack may involve more than one malicious forwarding device (see Figure 1). Compound attacks are challenging for solutions such as SPHINX 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, and this will trigger an alarm. We report that in our simulations a total of 108 attacks involved more than one malicious forwarding devices. Specifically, 35% of these involved four malicious forwarding devices, 25% six forwarding device, 40% nine forwarding devices. In real-world scenarios, an attacker who has taken over nine forwarding devices of a network is a strong adversary. Specifically, in AARNet Setup this means that the 75% of forwarding devices are compromised (this is a condition not supported by [11] requiring the majority of forwarding devices to be non-malicious).

9.4 Accuracy & Detection Time

We measured WedgeTail's detection accuracy in respect to A) Successful detection rate against attacks implanted in our simulated networks, B) Successful detection rate under network congestion leading to packet loss C) Successful application of pre-defined policies against malicious forwarding devices.

For A, we implanted attacks as specified in §9.3 over our simulated networks. We then used WedgeTail to measure the absolute time taken to detect the faults. The detection time is defined as the time taken to raise an alarm from the instant a malicious packet is routed over the network by a forwarding device. We report that all of the 500 attacks implanted in the networks were successfully detected by WedgeTail. The distribution of attacks over the networks was as following: 50 were over AARNet Setup, 250 over Zib54 Setup and 200 over Sprint Setup. Essentially, AARNet Setup was part of our feasibility study stage and Zib54, and Sprint Setups serve to prove the practicality of WedgeTail in real-world. 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 about 54 second with a standard deviation of 12 seconds. For Zib54 Setup, the average detection time is about 705 second with a standard deviation of 80 seconds. For Sprint Setup, the average detection time is 5600 second with a standard 730 second. Moreover, the average detection times were not affected in the presence of Compound Attacks (see §9.3). The latter is, in fact, expected given that the detection algorithm entails analyzing each and every forwarding device and the response engine is not triggered until the end of a full scan.

The aforementioned performance metrics show that WedgeTail's detection time scales well as the network size increases. The detection time of attacks over network snapshots is also acceptable. In other words, for a network administrator to be able to detect and locate malicious forwarding device after about 90 minutes without defining any policies or manual investigation is quite satisfactory.

Detection Delay	Accuracy	False Positive	False Negative
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.

For B, we added random congestion to the simulated networks, which resulted in packet drops at various points in the simulated networks. 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 2 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. The results prove that packet loss due to congestion is not a prohibitive factor for our system. WedgeTail is now only able to distinguish between packet drops due to congestion and maliciousness. We acknowledge that we have not measured the impact of congestion on successful malicious forwarding device localization and we leave further investigation for our future work.

Regarding C, we report that WedgeTail has matched policies with the threats and applied the actions specified in the policies for all attacks.

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.

1. Target Identification: Figure 8 illustrated the target identification time with respect to the number of forwarding devices that exist in networks. The algorithm takes approx. 18 seconds to identify the target forwarding devices in AARNet Setup with 400 trajectories. This number increases to up to 12 minutes for Sprint Setup, where there are approx. 640000 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 both of the control plane and data plane in SDNs.

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.

4. Resource Utilization: We observe WedgeTail reaches a maximum CPU usage of approx. 15% and memory usage of approx. 18%. The CPU usage is mainly associated with target identification and packet tracking components of WedgeTail.

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 [28], which takes a snapshot of forwarding tables and analyze them for errors, and NetPlumber [18] 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 more than just evaluating rule sets (e.g. identifying scanning targets, tracking packets as they traverse the network, maintaining a network replica to remove trust from forwarding devices, etc.) with a reasonably added overhead.

10. THE GOOD, THE BAD AND THE UGLY

10.1 Why WedgeTail and SPHINX?

At this point, we would like to draw a clear line between network troubleshooting solutions and our work. Solutions such as [18, 20, 34], mainly, focus on the elimination of configuration conflicts, the avoidance of routing loops and black holes, the detection of policy inconsistency, and etc. However, even with a correct configuration, the forwarding devices may fail in execution due to bugs in switch software, conflicts, limited memory space. A simple failure to execution is itself worrisome but a malicious forwarding device is a serious threat to the network operators and associated hosts. Recently, threat from state actors and insiders are on the rise. With such strong adversaries, it is feasible to expect the attackers to exploit forwarding device vulnerabilities in the core of the networks to achieve their goals (e.g. surveillance, etc.). There are even simpler, and potentially more dangerous scenarios, when compromised forwarding devices are purposefully placed by insiders in such networks. Hence, we strongly believe networks require to have solutions built against such adversaries. One should note that these solutions should not be, mainly, measured regarding the detection time rather successful detection of all threats within a reasonable time. In summary, we regard WedgeTail as complementary to solutions provided for network troubleshooting and for this reason WedgeTail is built on top of the most robust proposals in the network troubleshooting domain including HSA [19].

10.2 Why WedgeTail?

Related works including [11], mainly, rely on administrator-defined policies for attack detection, are built against weaker adversarial settings and fail to detect certain types of attacks (see §9.2). Moreover, they do not discuss localization of malicious forwarding devices, imposes some overhead to network performance, cannot distinguish between malicious actions such as packet drop or delay and do not prioritize the inspection of forwarding devices.

10.3 Limitations and What's Next?

We evaluated WedgeTail over various network setups, configurations, and sizes equipped with different SDN controllers to prove its practicality under simulated environments closely matching real-world networks. 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

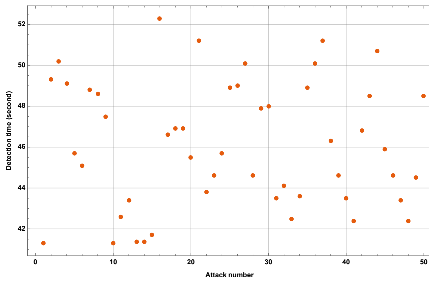


Figure 5: Attack detection in AARNet Setup (50 attacks).

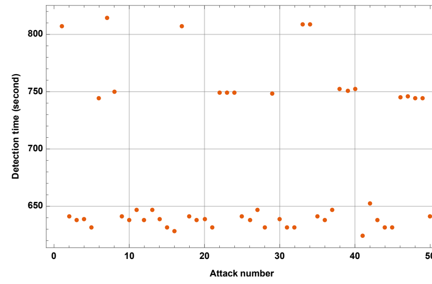


Figure 6: Attack detection in Zib54 Setup (50 attacks).

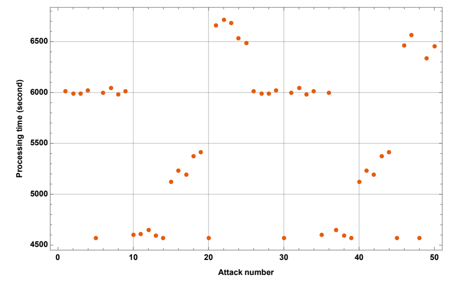


Figure 7: Attack detection in Sprint Setup (50 attacks).

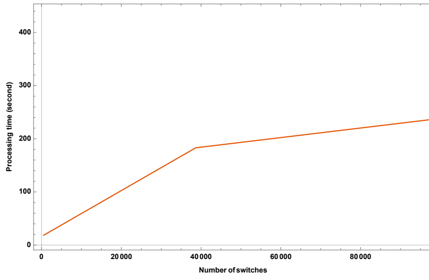


Figure 8: Target identification with respect to the number of forwarding devices.

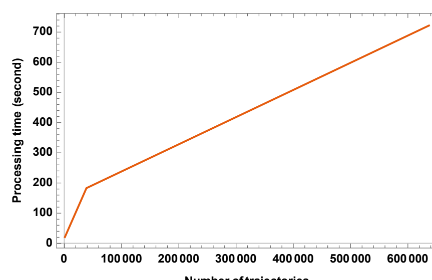


Figure 9: Target identification with respect to the number of trajectories.

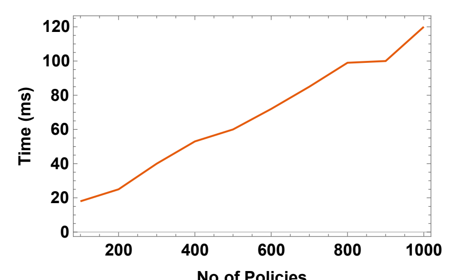


Figure 10: Average policy matching times with increasing policies.

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.

Another issue to point out is that our system analyzes snapshots and the stability of snapshots may be challenging [41] – as with all other similar offline systems proposed. 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

In the era of cyber-war, cyber-terrorism and with insider threats reportedly on the rise, it is to expect for attackers to exploit the vulnerabilities of the network core infrastructure to launch attacks against networks. Currently, Software Defined Networks (SDN) is regarded as the networks of the future. 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 [11] 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.

Acknowledgments

The authors would like to express their gratitude and appreciation to all the anonymous reviewers for their comments on the paper. Specifically, we are grateful to our Shepherd Dr. Cong Wang for his valuable feedback and assistance in improving the quality of this work. The first author also acknowledges the technical suggestions and recommendations of his former colleagues at Information Security Research Group of University College London (UCL).

12. REFERENCES

[1] Mausezahn. http://www.perihel.at/sec/mz/.

[2] Open Networking Foundation (ONF). https://www.opennetworking.org/.

[3] 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.

[4] G. Andrienko, N. Andrienko, S. Rinzivillo, M. Nanni, and D. Pedreschi. A visual analytics toolkit for cluster-based classification of mobility data. In *International Symposium on Spatial and Temporal Databases*, pages 432–435. Springer, 2009.

[5] G. Andrienko, N. Andrienko, S. Rinzivillo, M. Nanni, D. Pedreschi, and F. Giannotti. Interactive visual clustering of large collections of trajectories. In *Visual Analytics Science and Technology, 2009. VAST 2009. IEEE Symposium on*, pages 3–10. IEEE, 2009.

[6] 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.

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

[8] 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.

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

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

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

[12] 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.

[13] 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.

[14] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi. Trajectory pattern mining. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 330–339. ACM, 2007.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] J.-G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *2008 IEEE 24th International Conference on Data Engineering*, pages 140–149. IEEE, 2008.

[28] 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.

[29] 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.

[30] S. Meloni, J. Gómez-Gardenes, V. Latora, and Y. Moreno. Scaling breakdown in flow fluctuations on complex networks. *Physical review letters*, 100(20):208701, 2008.

[31] 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.

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

[33] 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*.

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

[35] 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.

[36] 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.

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

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

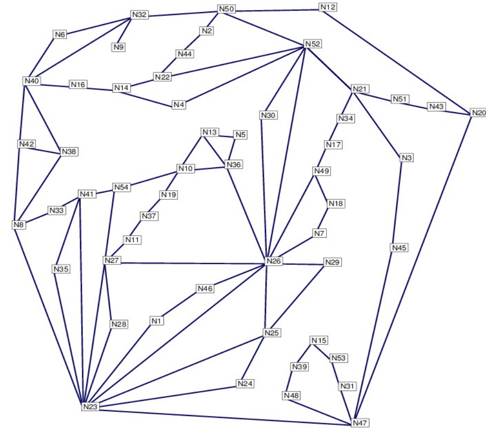
[44] Y. J. Zhu and L. Jacob. On making tcp robust against spurious retransmissions. *Computer communications*, 28(1):25–36, 2005.

APPENDIX

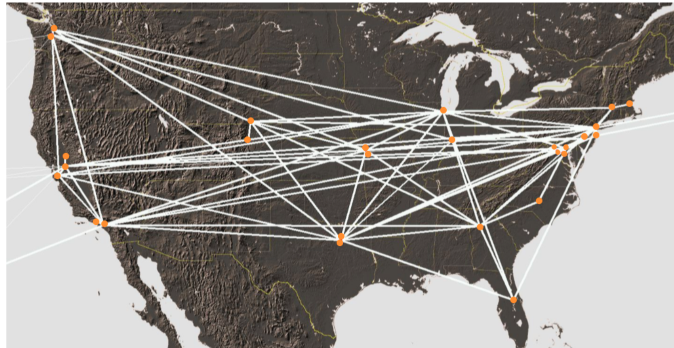
A. NETWORK TOPOLOGIES

For the sake of completeness, we include a representation of network topologies used in our evaluations in Figure 11. Figure 14a shows the topology used in AARNet Setup. The Image and topology for this are extracted from the Internet Topology Zoo [23]. Figure 14b illustrates the topology used in Zib54 Setup. The image as well as topology are extracted from SNDlib [34]. Figure 14c shows the network topology used in Sprint Setup. In this setup, each node of the figure is constituted of multiple interconnected forwarding devices. Image and topology are extracted from [39].

a: AARNet network topology simulated in AARNet Setup.



b: Zib54 network topology simulated in Zib54 Setup.



c: Backbone topology of Sprint simulated in Sprint Setup.

Figure 11: Network Topologies used in WedgeTail Evaluations