Towards Better Kernel and Network Monitoring of Software Actions

by

Yunsen Lei

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

Yunsen Lei

May 2020

APPROVED:

Professor Craig A. Shue, Thesis Advisor

Professor Lorenzo De Carli, Thesis Reader

Professor Craig E. Wills, Department Head

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to my advisor Professor Craig Shue for his support and guidance through my graduate study. He offered me a chance to conduct research with him the first semester I got to WPI. Without him, I would not have begun the journey of becoming a security researcher. The invaluable research skills that I learned from him is instrumental in making me a better researcher. The patience he placed in me and encouragement from him made me confident in solving any challenges during my research. I also want to thank Professor Lorenzo De Carli for his valuable feedback and time spent to help me make this thesis a success. I would also like to give special thanks to Beckley Schowalter, who helped me with the proofreading and gave many suggestions to improve the writing. I would like to express my gratitude to the National Science Foundation for their generous funding. This material is based upon work supported by the National Science Foundation under Grant Nos. 1422180 and 1814402. Finally, I would like to thank my parents. I thank them for their unconditional love. Their support on my education and the precious values they taught me have made me what I am today.

Abstract

Monitoring software actions is one of the most studied approaches to help security researchers understand how software interacts with the system or network. In many cases, monitoring is an important component to help detect attacks that use software vulnerabilities as a vector to compromise endpoints. Attacks are becoming more sophisticated and network use is growing dramatically. Both host-based and network-based monitoring are facing different challenges. A host-based approach has more insight into software's actions but puts itself at the risk of compromise. When deployed on the server endpoint, the lack of separation between different clients only further complicates the monitoring scope. Compared to network-based approaches, host-based monitoring usually loses control of a software's network trace once the network packet leaves the endpoint. On the other hand, network-based monitoring usually has full control of a software's network packets but confronts scalability problems as the network grows. This thesis focuses on the limitations of the current monitoring approaches and technologies and proposes different solutions to mitigate the current problem.

For software-defined networking, we design and implement a host-based SDN system that achieves the same forwarding path control and packet rewriting functionality as a switch-based SDN. Our implementation empower the host-based SDN with more control in the network even without using any SDN-enabled middleboxes, allowing SDN adoption in large-scale deployments. We further corroborate flow reports from different host SDN agents to address the endpoint compromise problem. On the server endpoint, we leverage containers as a light-weight environment to separate different clients and build monitoring infrastructures to narrow down the monitoring scope that have the potential to facilitate further forensic analysis.

Keywords— Security monitoring, Software-defined networking, Containerization

Contents

1 Introduction **1**

2 Background and Related Work **5**

2.1 Software-Defined Networking and its Application 5

2.2 Software-Defined Networking Limitations 6

2.3 Host-Based SDN 6

2.4 Spanning Trees and Virtual LAN 7

2.5 Kernel Monitoring 7

2.6 Detecting Compromises on Endpoints 8

2.7 Containerization and Virtualization 8

3 Host-Based SDN **10**

3.1 Required Functionality in Host-Based SDNs 10

3.2 Design Goals 11

3.3 Strategic Preallocation of VLAN Spanning Trees 12

3.4 Flow Header Rewriting at the Host SDN Agent 15

3.5 Implementation 15

3.5.1 SDN Controller 15

3.5.2 Host SDN Agent 16

3.6 Evaluation 16

3.6.1 Experiment Setup 17

3.6.2 Packet Header Modifications 18

3.6.3 Evaluating Arbitrary Forwarding Path Functionality 19

3.6.4 Impact on Host Flow Table Size 21

3.6.5 Elevation comparison with switch-based SDN agent 22

3.6.6 Impact on Legacy Switch Table Size 22

3.7 Discussion and Conclusion 24

4 Detecting Host SDN Agent Compromise **25**

4.1 Corroborated Host-Based SDN Agent Enforcement 25

4.1.1 System Overview and Threat Model 25

4.1.2 Corroborated Sensing Deployment Scenarios 27

4.2 Uncorroborated Data in Host SDN Agent 29

4.3 Implementing the CHOSE System 30

4.3.1 SDN Controller Customization . . . 30
4.4 Evaluating the Security and Performance of CHOSE . . . 31
4.4.1 Experiment Setup . . . 31
4.4.2 Performance Evaluation . . . 32
4.4.3 Round Trip Timings . . . 33
4.5 Security Evaluation . . . 33
4.6 Discussion . . . 34

5 Monitoring on Server Endpoint **36**
5.1 Single-Use Server Model . . . 36
5.2 Client-to-SuS Middlebox . . . 37
5.3 Container Auditing . . . 38
5.3.1 Overview . . . 39
5.3.2 Container Startup . . . 39
5.3.3 Container Auditing Design . . . 39
5.3.4 Container Auditing Implementation . . . 41
5.3.5 Evaluation . . . 42
5.3.6 Discussion . . . 43

6 Future Work and Conclusion **44**

References **45**

List of Figures

1 Comparison between switch-based SDN and host-based SDN . . . 10
2 A network with 2 VLANs configured so the links of each VLAN consists of different spanning trees . . . 12
3 Experiment network topology and VLAN configuration . . . 17
4 No packet modification . . . 19
5 VLAN tagging . . . 19
6 DSCP field modification . . . 19
7 Round trip time with different forwarding path . . . 20
8 Round trip time under different flow table size . . . 21
9 Round trip time comparison between host agent and switch agent . . . 22
10 An example enterprise network with SDN agents on each end-point. . . . 26

11 When both endpoints run an SDN agent, if either is uncompromised, that uncompromised agent will alert the central coordinator of inconsistencies via its `PacketIn` data. . 27
12 When an SDN switch is on the network path, the controller receives `PacketIn` data that allows it to identify which endpoint, if any, is faulty. 28
13 When one of the hosts is compromised (shaded in black), the controller will notice a discrepancy when receiving a `PacketIn` from the non-compromised host (shaded in gray). 28
14 If both hosts collude (shaded in black), only a middlebox or an SDN switch between the hosts can be used to detect the malicious flows. 29
15 Round-trip time of serial connections across 500 trials. 33
16 Tree structure to determine if a process is a in-container process 41
17 The components in the container audit program 42

List of Tables

1 Host-Based SDN Components and Functionality 18
2 SDN switch TCAM table comparison 21
3 Number of connections allowed and denied by scenario. 34

1 Introduction

Software, which was created to facilitate the use of a computer system, has now become the cause of many security problems. Due to the natural complexity in its design, software usually contains vulnerabilities that give attackers the chance to corrupt the entire system. Sometimes, such corruption is achieved without leveraging a software vulnerability: an email with a malware download link, that deceives the user into clicking it can also corrupt the system.

To protect our systems against these threats, researchers have proposed different approaches. Some have tried to run a untrusted system in an isolated environment through virtualization [22]. Others have added instrumentation at a binary level to capture the behavior of software [2] or tried to distinguish a normal software from a malicious one through learning-based algorithm [3]. Though they rely on different assumptions or threat models, these approaches share one thing in common: they try to characterize the software's behavior.

To gain knowledge of software's behavior, a widely-used method is to monitor software actions. Monitoring systems deployed within the host can provide more insight into software activities. By vetting network activities or auditing system call traces in the kernel, the security system gains internal view of the network stack and understands software actions at a functional level [34]. While this approach provides insight, the cost of this increased visibility is that the monitoring system is as likely to be compromised as the software being monitored. Most endpoint monitoring systems rely on a trustworthy kernel space. In the presence of a kernel compromise, these monitoring systems may use anti-circumvention techniques to hinder their removal or sabotage. However, the monitoring can be manipulated to provide faulty information about a software's action. In addition to host-based monitoring, there are monitoring systems that are deployed in the network, usually on a network middlebox (where they have more attack resistance). Compared to host-based monitoring approach, these network middleboxes (usually under control of the network operator) can also act as part of the defense-in-depth component to form an overlapping control that rejects a software's abnormal network activity even when endpoint security system is compromised.

To gain advantages above, visibility is sacrificed. Network monitoring system usually have to monitor multiple sources which can result in scalability problems they are used in large networks. For both network-based and host-based monitoring, scalability, visibility and attack resistance are three different metrics that hard to balance. Previous work has proposed virtual machine introspection [23], which pulls the monitoring system outside the monitored system for greater attacker resistance. It's the virtual machine monitor (VMM) on the host machine to retain the same visibility as a host-based method. Because of the strong isolation between the host and guest system, the semantic gap between the host and guest machine makes it hard to extract meaningful information [17]. For

network-based monitoring, some works have focused on its scalability problems by using a decentralized approach [54] at the cost of higher management overhead. Other work has tried to extract information about the endpoint software from the network traces [3] using learning-based algorithm to compensate for decreased visibility. These approaches mitigate the visibility problem but are subject to adversary learning, in which the attacker deceives the algorithm to a provide false fingerprint for the network traffic [5].

Monitoring software actions on the server endpoint raises additional challenges. A server application is usually a piece of software that allows multiple users to share the service at the same time. When multiple users access the server simultaneously, concurrent streams are generated from each user that cause the monitoring system only to monitor an intertwined network flow and execution traces. This significantly increases the difficulty of analyzing the monitoring data and makes it even harder to associate a server's action with individual users. To facilitate analyzing the server's action and isolating different users, previous work [52] created a one-to-one client-server model using virtual machines. But virtual machines they use are slow and require significant resources, which hinders the scalability.

The many problems mentioned above cannot be addressed by a single solution. This thesis focuses on a subset of the problems in turn. We propose to fuse theses solutions into a comprehensive system.

In the first direction, we focus on software-defined networking (SDN). The design paradigm of SDN enables flexible control of the network devices. Ever since it has been proposed, researchers have been taking advantage of this feature to develop different applications in various domains. For this part, we focus only on SDN's ability to monitor network and perform traffic engineering. When used as a network monitoring tool, SDN allows the user to specify flexible policies that achieve traffic monitoring at different granularity. Its traffic enginnering feature can be used to control the path of a network flow, such as forcing a packet to go through multiple layers of defences and allowing network operators to practice a defense-in-depth strategy. SDN also allows network middleboxes to perform packet header rewriting, allowing defenders to perform obfuscation that achieves a networked moving target defence (MTD) [32]. Despite SDNs provide great promise, previous works have shown that the SDN approach has scalability problems when deployed in a large scale network [50]. To address this problem, prior works that have tried moving SDN agent into an end-host [53, 43]. This transaction achieved better scalability and is able to provide more informed flow reports. But these efforts have not achieved SDN's ability to control the packet path and traffic engineering, so they have not fully realized SDN's ability to facilitate network and endpoint security. In Section 3, we show that a host-based SDN can achieve the same functionality as a switch-based SDN. We recognize that when moving SDN into the host, it faces the same security challenge as other host-based monitoring systems. In section 4, we show that flow reports from multiple host SDN agents can be corroborated to help the SDN controller detect a compromised agent. Such detection mechanisms allow the controller to gain authentic flow

reports about the software's action in the end-host and provides security guarantees for the host-based SDN system.

In the second direction, we focus on monitoring and logging the software's execution data on a server endpoint. We leverage a single-use server model to build monitoring infrastructures that untwine and forward different users' requests to different server instances. For each server instance, we leverage the Linux auditing system [26] to log important system calls on a per-server instance basis. In the single-use server model, each server instance runs in a tailored container environment and is associated with a specific user during a normal session. Because different users only access their own server instance, our monitoring infrastructure is able to get a clean view of user's requests and record the server's action for that specific user. Such monitoring mechanisms can separate the user's forensic evidence and greatly reduce the volume of monitoring data needed to conduct any post-attack analysis. In Section 5, we first give an overview of the single-use server model that we developed and focus on the design and implementation of the monitoring infrastructure that work withs the model.

By exploring these directions, we make the following contribution:

- **Building SDN Infrastructure for a Legacy Network:** We leverage features in managed switches, such as the support for multiple spanning trees in different virtual local area networks (VLANs), to manipulate forwarding paths on legacy switches. We create an OpenFlow-compatible SDN controller to build custom topologies and configure the legacy switches. Our controller supports OpenFlow clients and manipulates state to allow arbitrary forwarding paths.

- **Design and Implementation an Endpoint SDN agent:** Given the popularity in enterprise networks, we implement an SDN agent in the Microsoft Windows operating system. The agent uses a kernel-mode network driver to implement the SDN controllers orders and to rewrite packets and perform flow path control.

- **Corroborated Host-based SDN Enforcement system**: In our **C**orroborated **Ho**st-based **S**DN **E**nforcement (CHOSE) system, an endpoint sensor reports flow and contextual data (e.g., originating user and application) for each new network connection. We design and implement a SDN controller that correlate host agents' report. When switch-based SDN agents are involved in the communication path, its report will be used as part of the verification as well.

- **Evaluating Traffic Engineering, Performance, Scalability and Security:** We first evaluate our host-based SDN in an experimental network environment and showed that it can implement flow rules, QoS field manipulation, and path selection equivalent to switch-based SDNs with only minimal overheads (Section 3.6). We then evaluate the CHOSE system and show that it can detect and block flows with inaccuracies that indicate a compromised host SDN agent.

We evaluate our system and show that when a communication path involve two host SDN agent and a single switch SDN agent, our controller can verify the flow information and allow the flow to establish within 20ms for about 90% of the flows.

- **Implementing Container Monitoring Infrastructure:** Based on the single-user server model, we design and implement two monitoring components. The first component is a proxy middlebox that serves as a demultiplexer for the single-use server model. By using a cookie-based method, our proxy is able to associate each user sessions with its assigned container. The second component is a container system call auditing system. Since the original Linux audit system cannot be namespaced [18], it cannot be used directly to audit system calls on a per-container basis. Our implementation relies on the `kauditd` to provide a host-wide view of the system call and `netlink` socket to receive container process event. When combining the two component together, we are able to separately log each container's system call on the host machine.

2 Background and Related Work

In this section, we introduce the basic concepts involved in this work and discuss some related works.

2.1 Software-Defined Networking and its Application

In the software-defined networking paradigm, control-plane decisions are separated from the underlying hardware that performs packet forwarding and modification. The OpenFlow protocol [38] provides an API for a logically-centralized controller to interact with a set of packet forwarding devices, which are often network switches. Because it uses a standard protocol, devices from difference manufacturers can be managed using the same language. In OpenFlow, a switch sends a `PacketIn` packet to an OpenFlow controller whenever it encounters a packet whose fields are not a match for any of the switch's cached rules. When issuing the `PacketIn` request, the switch includes a copy of the associated packet. The controller consults its policy to determine the appropriate action. The controller may optionally create a `FlowMod` packet to order the switch to store a new rule or update the existing rules with match criteria corresponding to the flow along with an action the switch should take on future matching packets. Finally, the controller issues a `PacketOut` message that telling the controller what to do the with the packet contained in the `PacketIn` message.

The OpenFlow protocol allows a controller to essentially treat each SDN switch as a configurable rule cache. A controller can push coarse-grain rules, which contain wildcards for various flow headers, to allow a switch to operate with few `PacketIn` elevation requests. Alternatively, a switch can use fine-grained rules, which typically specify a fixed flow tuple (i.e., IP_{source}, $\text{IP}_{dest.}$, transport protocol, port_{source}, $\text{port}_{dest.}$) that only matches a single connection. This design paradigm and protocol implementation provide a structural framework for researchers to explore different applications.

The use of fine-grained rules can be attractive for security purposes because the resulting packet elevations give the OpenFlow controller detailed visibility into the communication occurring on the network. This empowers the controller to act as a network-wide flow-based access controller. Based on such features, researchers have proposed different approaches that leverages SDN for network monitoring. OrchSec [63] is a monitoring system that detects attack traffic by incorporating the functionality of a network monitor to signal initial signs of an attack to the SDN controller and dropping the attack traffic. Also leveraging the visibility of network flows and a global view of the entire network, Raumer et al. have proposed MonSamp [47], which is an SDN framework that performs a sample-based flow monitoring that satisfies the requirement for quality of service (QoS). Their approach samples only a subset of traffic from SDN switches to avoid the problem of packet drops that result from over-utilizing monitoring links. To provide an SDN-based solution for multimedia delivery, Egilmez et al. proposed OpenQoS [19], an SDN controller design that groups network traffic into multimedia flows

and data flows. It place multimedia flows on QoS-guaranteed routes while the data flows remain on their traditional shortest-path. Based on the monitoring ability of SDN, there are also efforts that leverage SDN to denial of service mitigation [41, 62] and intrusion detection [51].

2.2 Software-Defined Networking Limitations

Research depends on SDN's ability to provide network monitoring to allow the controller to make different decisions based on the flow information it receives from SDN agents. Different levels of granularity can be adopted to control which packets are elevated to the controller. Previous work found that fine-grained rules in OpenFlow come at a cost [14]. While MAC and VLAN entries can be managed in SRAM, SDN rules involving other fields must be stored in ternary content addressable memory (TCAM). TCAM memory is expensive, both financially and in terms of energy consumption. Some switches can store around 2,000 entries, while others, such as the Dell PowerConnect 8132F, only store 750 OpenFlow rules [35]. OpenFlow-enabled switches also have a price premium compared to similar-capacity traditional managed switches. To address the limitation imposed by the use of TCAM, Wen et al. proposed RuleTris [59], an SDN flow table update optimization framework that leverages dependency graphs to minimize the update delay. When combined with their dependency preserving algorithm, RuleTris can achieve a 15 ms end-to-end per-rule update latency. TCAM's limited capacity is another problem for TCAM. To efficiently make use of the TCAM, Katta et al. proposed CacheFlow [33]. Their system caches the most popular rules in the TCAM. To handle table miss caused by other unpopular rules, they rely on a software approach that leverage a ”splicing” technique that break the long dependency chain and create a few new rules that covers those unpopular rules. Their method proved to be effective in handling table misses.

2.3 Host-Based SDN

While OpenFlow was originally designed for use with physical hardware in network switches, one of the more popular OpenFlow implementations is in software. Open vSwitch (OVS) [45] is often used on virtual machine (VM) hypervisors to provide SDN functionality between VMs. In the Scotch approach, Wang et al. [58] proposed using OVS to enhance the scalability of fine-grained flows by using OVS on VM hypervisors. As shown in previous work, OVS tends to have better performance than physical switches [29], but such performance advantage is gained by running the agent on a server and the host in the VM. OVS is not designed just for the a single physical endpoint.

Even with OVS, the SDN agent is still regarded as a network SDN agent. Although this network-based approaches provide more attack resistance, but the visibility may be insufficient. To further increase the visibility, Taylor et al. [53] proposed a host-based SDN that monitors the activity on the

end-host in addition to the network flow. Najd and Shue [43] transformed Taylor's host-based SDN into an OpenFlow-compatible implementation that could complete a flow elevation to a controller in less than 9 milliseconds.

2.4 Spanning Trees and Virtual LAN

A virtual local area networks (VLANs) create logical network segments, each with their own broadcast domains with on top of a physical network. Hosts in different VLANs cannot directly communicate without traversing a middlebox or router that spans the VLANs. VLANs can span physical switches by tagging each Ethernet frame with a VLAN ID. Each physical switch interface is configured as either an access port, which accepts only a single VLAN, or as a trunk port, which can carry traffic from multiple VLANs. In a VLAN-enabled network, switches maintain a MAC address table which stores entries containing VLAN identifiers, MAC addresses, interface ports, and aging timers. The switch uses both the VLAN identifier and the MAC address to determine which interface port to use for each packet. Packets without a tag are assigned to a default VLAN identifier associated with the interface.

Since each VLAN has its own set of interface port restrictions, each VLAN can have its own spanning tree. The multiple spanning tree protocol (MSTP) allows interface ports to trunk multiple VLANs to create multiple logical links for the underlying physical links. Despite the physical infrastructure having loops, each VLAN spanning tree can selectively disable ports to create a tree. The multiple spanning tree approach can address the scalability concerns in Ethernet. To build scalable Ethernet for the data center, Mogul et al. proposed SPAIN [42], which consists of a host driver program that randomly chooses a path from a set of usable spanning trees. They can achieve high throughput and fault tolerance when a forwarding path fails to deliver packets. The PAST [50] approach uses a per-address spanning tree rather than a per-VLAN spanning tree, which requires entries to be stored in the TCAM table of a SDN switch.

2.5 Kernel Monitoring

The kernel monitoring approach can get system call traces for software. By analyzing the temporal ordering of the system calls, the security system can learn characteristics that reveal how software operates at a functional level. This information can be used to define the normal behavior of the software. Earlier work leverages the visibility in kernel space to monitor software's behavior and secure the user-space application. Mitchem et al. [40] use a loadable kernel module that monitors system calls and tries to detect malicious behavior. This approach requires the user to be trusted to not disable the module. In the cloud computing environment, Thu Yein et al. proposed a kernel monitoring approach [60] that combines the system call monitoring in the guest OS kernel with SVM-

based external monitoring on the host machine. Their approach only monitors a subset of system calls. To prevent rootkits from modifying the system call table in the guest kernel, they use system call hashing that extracts a copy of the system call table upon installation.

2.6 Detecting Compromises on Endpoints

Once a host is compromised, an attacker may attempt to conceal the compromise in order to remain persistent or to spread laterally across the network. When trust assumptions, such as a trustworthy OS or kernel space, are violated, attackers can deactivate a host's defenses. For example, malware has been found to deactivate anti-virus [39] to evade detection. Attackers may also disguise their traffic, using mimicry techniques [57] to appear legitimate.

To relax assumptions about a trustworthy OS, trusted hardware, such as trusted platform modules (TPMs) [55] or secure co-processors [48], can be used to provide attestations. These approaches tend to suffer from fragility to minor changes (when a static root of trust is used) [10] or from classic time-of-check-time-of-use (TOCTOU) issues [9] (when a dynamic root of trust is used).

Our CHOSE approach differs from the above methods. We try to detect security systems that provide inaccurate data, or omit data, by comparing their monitoring data with data from other security systems on the network. This distributed monitoring approach can highlight attacks even without special trusted hardware.

2.7 Containerization and Virtualization

A monitoring system often shares the same operating systems with the monitored software. Kernel monitoring approaches rely on the kernel's functionality to be intact. Otherwise they cannot provide any authentic monitoring data. To make the monitoring system more attack resistant, other works [6, 30, 31] tries to break the in-host model by moving the security system to an isolated environment while keeping some visibility to the monitored system. Tal et al. proposed a VM introspection-based intrusion detection system [23] that breaks the traditional in-host monitoring model. In their work, they use virtual machine monitor technology, which gives them access to the hardware state information about the virtual machine. By interpreting these hardware events at the OS-level, their approach allows them to define IDS policies. By controlling and monitoring the guest OS at both the hardware and the software level, they add attack resistance. However, this approach introduces semantic gaps where the the security software needs to infer the activities that happened in the VM. Another virtualization technology which has gained popularity is containerization. Unlike virtual machines, containerization is achieved by leveraging a sequence of security enhancement functions and isolation technologies that are already used in the host system. Therefore, such isolation is light-weight and does not require an

environment on a separate operating system. Among the various container technologies, Docker is one of the most popular. Using AppArmor Mandatory Access Control (MAC) [56], a Docker container prevents links to secure sensitive directory such as `/etc` and `/sys`. The use of `cgroup` and `namespace` further partition and limit the accessible resource from within the container. In addition, Docker adopts a per-container level `seccomp` filtering that whitelist system calls [4]. Such environments provide the isolation we need to run multiple server instances that each only serve a single user. In our work, we designed a single-use server model where the monitored server software is running in a container and the monitoring mechanism operates from outside the container. Using the isolation provided by the container, our approach can preserve attack resistance while being able to directly observe software actions.

3 Host-Based SDN

SDNs offer a lot security benefits, but the limitations of current SDNs impede their adoption and application on large scale. For Switch-based SDNs, when a SDN agent is deployed on the network perimeter, it foresees the network event before the inbound traffic reach the endpoint, allowing network operators to stay ahead of security threat. When deployed within a network, it observes network activities between the endpoints and checks any sign of internal threat. Its ability to control the packet's forwarding path enables flexible network isolation. But to use switch-based SDN, the most obvious limitation is cost: the physical switch hardware must support SDN and this may require upgrading network infrastructure which lead to both capital and installation costs [8]. Another limitation is that hardware components on the SDN switch hinder the use of fine-grained rules to use in large network. Host-based SDNs provide more detailed flow reports to the network operator and can scale better than the switch-based SDN. However they have not achived the same features as switch-based SDN.

Given the benefits and limitations of both types of SDN, we explore the the possibility of combining two types of SDN and ask the following research questions: *Can host-based SDNs achieve all the original goals of the SDN paradigm? What are the costs of performing switch-based SDN functions on a host-based SDN?*

3.1 Required Functionality in Host-Based SDNs

We first examine the functionality requirements for host-based SDNs to achieve feature parity with switch-based SDN. We then describe the design and implementation in achieving these functionality. In the SDN design paradigm, the SDN agent can be located in both a physical switch or in the host. As showen in Figure 1, the location of the SDN agent does not affect the original design pattern. The controller can still have a global view of the network. Through protocols like OpenFlow, the controller can manage each host agent the same way as it does for the switch agent. Such a simple shift may

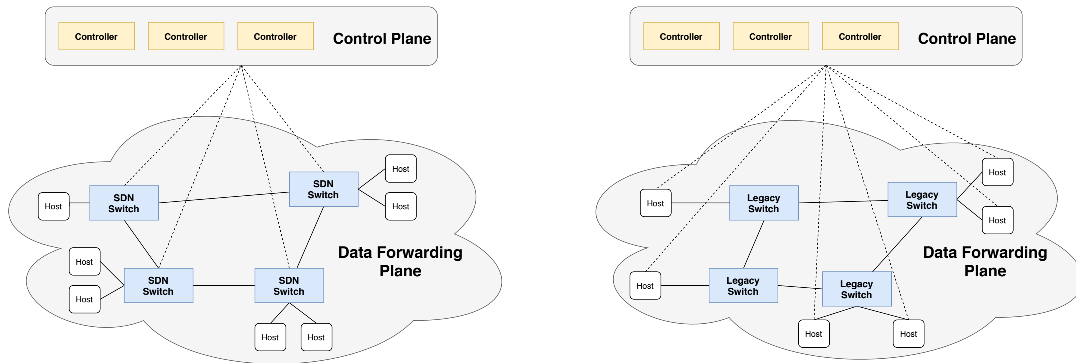

Figure 1: Comparison between switch-based SDN and host-based SDN

seems easy to accomplish, but some unsolved problems remains. SDN switches have multiple interface ports and can prioritize queued traffic or forward packets through arbitrary interface ports. These capabilities are key for traffic engineering and quality of service. However, it is unclear if SDN agents on hosts can achieve similar functionality on legacy switches. To be considered equivalent, a host-based agent would need to be able to achieve the following requirements:

- **Influence the Forwarding Path:** While a host-based SDN cannot specify the forwarding path of a packet, it can influence how switches will treat a packet. With carefully preallocated VLAN spanning trees, a host-based SDN agent may be able to select a VLAN to determine the path.
- **Rewrite Packet Header:** The SDN agent must be able to rewrite packet headers. Such function is necessary to achieve some SDN applications. In terms of quality of service, rewriting packet headers can trigger prioritization and QoS features when host agents are used with managed switches. When implemented with proper functions, it also allow the host agent to perform packet transformations (e.g., NAT) or packet tunneling protocols (e.g., MPLS).

Combined, these features would allow network operators to achieve the same goals as in switch-based SDN.

3.2 Design Goals

Our approach is designed to support SDN in a regular legacy network. Such networks can include data center network or even a large enterprise networks. These networks typically have managed switches that support VLANs and traffic engineering options. While networks can use unmanaged switches, these switches lack VLAN, traffic engineering, and even configurations like the spanning tree protocol that avoid link loops, limiting traffic to a single forwarding path between destinations. We thus design our approach with traditional, non-SDN managed switches and routers in mind.

Our approach is designed to require only minimal host configuration via a kernel network driver that can be automatically installed with standard software deployment tools. We further support legacy hosts that do not run the SDN software, such as printers or embedded devices, but note that the traffic engineering functionality may be unidirectional with legacy devices since only the SDN-enabled host would be able to use non-default VLANs or QoS fields.

Our traffic engineering is designed to support network path selection through a sequence of network middleboxes. Such middleboxes can be a firewall or IDS system. Further, the approach is designed to be scalable even when fine-grained match rules are in place.

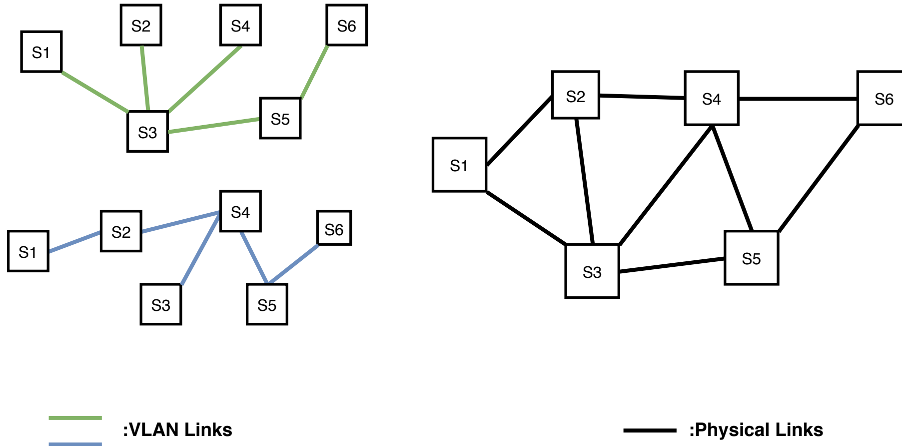

Figure 2: A network with 2 VLANs configured so the links of each VLAN consists of different spanning trees

3.3 Strategic Preallocation of VLAN Spanning Trees

We leverage VLANs and their support for multiple spanning trees to allow hosts to influence forwarding paths. As described in Section 2, both the VLAN identifier and destination MAC address are used to determine which output interface port to use when multiple VLANs are configured on the switch. When redundant physical links are present in the network, there are multiple spanning trees for that network as well. To enable different forwarding paths, the network can be strategically configured with VLANs so link between each configured interface port that is part of the spanning trees of the network. Figure 2 shows two spanning trees based on the same physical network topology. For each spanning tree, a different VLAN identifier is configured on the corresponding interfaces. When VLAN tagged packets are forwarded in this network, different VLAN identifiers will yield different forwarding paths. Using such method, when a host agent sends out packets with a different VLAN identifiers, it enables the agent to influence a packet's forwarding path. In this section, we discuss in detail how we strategically configure VLANs in the network.

Spanning Tree Enumeration Given a network represented by a graph, we can determine the total number of spanning trees, t, of the graph by calculating the determinant of the graph's Laplacian matrix. Specifically, for a complete graph with n vertices, $t = n^{n-2}$. If we choose to configure each spanning tree with a single VLAN, such an approach may require VLAN identifiers that exceed the maximum number of VLANs that can be used in a network (which is 4,094 in the IEEE 802.1Q standard [25]). Therefore, when choosing spanning trees, we only select a set of spanning tress that covers a set of alternative forwarding paths rather than the default forwarding paths (which is already covered by the default spanning tree of the network). Such alternative forwarding path can be selected by the user who runs an SDN controller that programmatically calculates the path.

Before discussing the detail of the algorithm we used, we first define some notation. Let $S =$

$\{s_1, s_2, ..., s_t\}$ be the set of spanning trees for a network $G = (V, E)$ where $|V(G)| = n$ and $|E(G)| = m$. Let $\mathcal{P} = \{p_1, p_2, ..., p_k\}$ be the set of k different user-selected paths.

To compute the set S, we use Winter's approach [61] to enumerate the all the spanning trees for $G = (V, E)$. That algorithm recursively finds a partition in the spanning tree space by determining whether or not the edges connecting the biggest labeled vertex and its adjacent vertices belong to the spanning tree. It contracts the biggest labeled vertex into its adjacent vertices if the edge belongs to the spanning tree and deletes the edge if it does not. Such a contraction process is repeated until there is only one node left. This algorithm has a time complexity of $O(n + m + nt)$. Simply listing all the spanning trees of a graph requires $O(nt)$ time.

Path Selection To get the set P, we consider cases in which an alternative forwarding path between two switches is needed. In our design, all the hosts on the network are members of a default VLAN. This allows basic communication without requiring any special SDN rules. Therefore, the user-selected path set, P, does not contain the default path between each pair of switches. The SDN agent thus needs to change the packets only for flows in which a non-default forwarding path is desired by the SDN controller. There are many situations in which an alternative forwarding path might be desired. A user-selected path can be QoS motivated routing path where an alternative forwarding path is selected to avoid a hop spot in the network. It could also be a security-motivated path, where all the traffic from a host is being forwarded to pass through a network firewall. The SDN controller can arbitrarily select a different path for each network flow.

We generalize such cases into a constrained shortest path problem. Given a set of ordered vertices, $V' = \{v_1, v_2, ..., v_q\} \in V$, which represent a set of switches along our forwarding path, let set Σ contain all the possible permutations $\sigma = \binom{v_1, v_2, \ldots, v_q}{\sigma(v_1), \sigma(v_2), \ldots, \sigma(v_q)}$ of V'. Given a pair of nodes (a, b) and the set Σ, Algorithm 1 will compute the shortest path between a and b that visits all the nodes in V'. Algorithm 1 first compute the all pair shortest path for the network represented by graph $G = (V, E)$ and stores the result in a two dimensional array d where $d[i][j]$ represent the shortest path between vertices i and j. In iteration k, the algorithm compute the shortest path between a and b visit vertices V' in a ordering that specified by σ_k.

We show the correctness of the algorithm via a proof by contradiction. First, we show that sp calculated in each iteration is indeed the shortest path from a to b that visits each vertex in V' in that specific order. Suppose that there exists $sp' < sp$, then one of the sub paths in sp' must have a shorter distance than the same sub path in sp, which contradicts the fact that each sub path is already the shortest path that calculated by the all pair shortest path algorithm. Since our algorithm always updates the $\delta(a, b)$ when encountering smaller sp, it produces the correct result at the end. When computing the shortest path that goes through a single network firewall, algorithm 1 can be reduced to the calculation $\delta(a, f) + \delta(f, b)$, where f represents the firewall node.

Algorithm 1: Shortest walk including required nodes

```
initialize δ(a, b) = ∞;
d = all_pair_shortest_path(G);
for each σ_k ∈ 𝒫 do
    sp = d[a][σ_k(v_1)] + Σ_{n=1}^{q−1} d[σ_k(v_n)][σ_k(v_{n+1})] + d[σ_k(v_q)][b];
    if sp < δ(a, b) then
        δ(a, b) = sp;
    else
        continue;
    end
end
return δ(a, b);
```

This approach allows us to construct a simple path between any two nodes within at least one spanning tree. If the controller decides to use a path that is not covered by any spanning tree, then the controller must first configure a spanning tree to cover that path. This can be done proactively, before a flow is created, or reactively, when a packet is elevated to the controller. This VLAN tree configuration is equivalent to the `FlowMod` rules in a switch-based SDN.

VLAN Spanning Tree Selection After computing the spanning tree set S and the path set P, we then apply our spanning tree selection algorithm to compute the set $S' \in S$ such that covers every path in P. From the above, we see that the set $P = \{p_1, p_2, ..., p_q\}$ is the universe path we want to cover. And for each spanning tree $s_i \in S$ it covers a subset of paths in P. We let $s_i = P_i$ denote the spanning tree s_i covers every path of $P_i \in P$. Therefore, to compute S' that contain the minimal number of spanning trees is a set cover problem which is NP-Complete. Therefore, to calculate S', we use a greedy algorithm that repeatedly add the spanning tree that covers the most paths in the remaining part of the set P.

Suppose the spanning tree set S and path set S are already calculated using algorithms mentioned above. And for each spanning tree $s_i \in S$, we calculate the set of path it can over denoted by P_i. Algorithm 2 describe the greedy algorithm that calculates S'.

Algorithm 2: Greedy spanning tree selection

```
initialize S' = ∅ , P = {p_1, p_2, ..., p_q};
while P ≠ ∅ do
    find s_i that covers the most paths in P;
    S' = S' + {s_i};
    P = P − P_i
end
```

This greedy approach is an approximation algorithm that produces a sub-optimal solution within a logarithm factor to the optimal solution. If the optimal solution needs r spanning trees, then

Algorithm 2 requires at most $r \ln q$ spanning trees (q being the total number of paths in set P). With this knowledge, the user can decide whether to use an exhaustive search to find the optimal solution if the number of required spanning trees exceeds the network configuration limit.

3.4 Flow Header Rewriting at the Host SDN Agent

Our host-based SDN agent has two parts: 1) a kernel network driver, which supports packet inspection and header rewriting, and 2) a service that handles the OpenFlow communication. When new flows are detected, the kernel driver sends the packet to the OpenFlow service for instruction. That service elevates the packet to the controller if there are no relevant rules cached locally.

3.5 Implementation

The host-based SDN system implementation centers around two components: the SDN controller and the SDN agent on the endpoint.

3.5.1 SDN Controller

The SDN controller must preconfigure the network with VLANs that each are associated with a different spanning tree based on the result of the spanning tree selection algorithm. It also needs to configure managed switches with QoS settings so the host agent can send labeled packets to trigger the functionality. The controller can configure these values remotely using an SSL/TLS, SSH, or Simple Network Management Protocol (SNMP) API supported by the network switches and routers.

To calculate the appropriate VLAN spanning trees, the SDN controller needs to know the network topology. This can be learned via routing and spanning tree protocols. However, in our implementation, we assume the topology graph is already available. Our controller reads the topology graph represented in the `dot` [24] language. The `dot` language can specify the relationship between different vertices along with attributes for each edge and vertex in the graph. In our implementation, we include interface ports and network firewall information in the `dot` file.

Our SDN controller is implemented in C++. To handle communications with the host agent, we use the `botan` library to handle the TLS encryption and decryption. To parse the `dot` file, we leverage the functions in the `cgraph` library. Also, we implement the controller to use OpenFlow 1.0 [20] standard. When a VLAN tagging is required for the flow, the controller uses the `SetVLANVID` action in its `PacketOut` and `FlowMod` packet. The same action can also be used to set the PCP filed within the 802.1q filed to trigger QoS. To alter the DSCP field in the IPv4 header, the controller uses the `SetNWTos` action. We do notice that OpenFlow1.0 only supports a limited number of `SetField` actions. To enable more flexible packet header rewriting, a `SetField` action can be used (defined in

OpenFlow 1.3 [21]), which leveragesf a bit mask approach to support modification of multiple field in one action.

3.5.2 Host SDN Agent

Given the popularity of Microsoft Windows in enterprise networks, we implement the host SDN agent as a Windows kernel network driver that can be easily installed via software deployment tools without requiring end-user configuration.

In OpenFlow, an SDN agent typically only elevates the first packet of a flow to the controller unless instructed otherwise. We thus need a mechanism to identify new flows when they are created, elevate the first packet to the controller, and queue subsequent packets until a response is received from the controller. To achieve this goal, we leverage the Windows Filtering Platform (WFP) [13], a packet filtering engine that allows us to inspect and modify packets at different layers of network stack. To identify the creation of new flows, we register so-called callout functions with layers of the special Windows Application Layer Enforcement (ALE) group to monitor socket operations. ALE is a set of layers that trigger on the packets associated with the `connect` and `accept` system calls in TCP. For UDP flows, ALE layers trigger on the first packet that is sent to, or received from, a unique remote entity. Using this functionality, we are able to elevate packets on a per-connection or per-socket level.

The controller communication module is implemented as a user-space administrative service that implements the OpenFlow protocol. The communication with the controller is authenticated and encrypted. To elevate a packet to the controller, the service first receives the packet from the kernel driver and embeds it into an OpenFlow `PacketIn`. Upon receiving response from the controller, the service first decrypts and verifies the packet before delivering the OpenFlow packet to the kernel driver for processing.

The kernel driver implements the controller's instructions for the flow. These instructions typically involve packet header modifications, such as setting fields in the IPv4 header or Ethernet VLAN headers. To modify a packet efficiently, we adopt an in-line packet modification approach. We use callouts in the WFP engine to intercept the packet's header at different layers in the network stack. In our implementation, we support packet header modifications from the link layer up through the transport layer. We offload checksum recalculation from header modifications to the the NIC along with VLAN tag modifications, yielding high performance.

3.6 Evaluation

In our evaluation, we explore the effectiveness of our approach at achieving its traffic engineering goals along with the overhead of implementing these techniques in a host-based SDN agent.

3.6.1 Experiment Setup

To test our system, we build a full-mesh network topology of four switches, as shown on the left in Figure 3. We flash four consumer-grade TP-Link Archer C7 routers with the OpenWrt [46] firmware to act as our managed switches. Each switch has its wireless and routing functionality disabled.

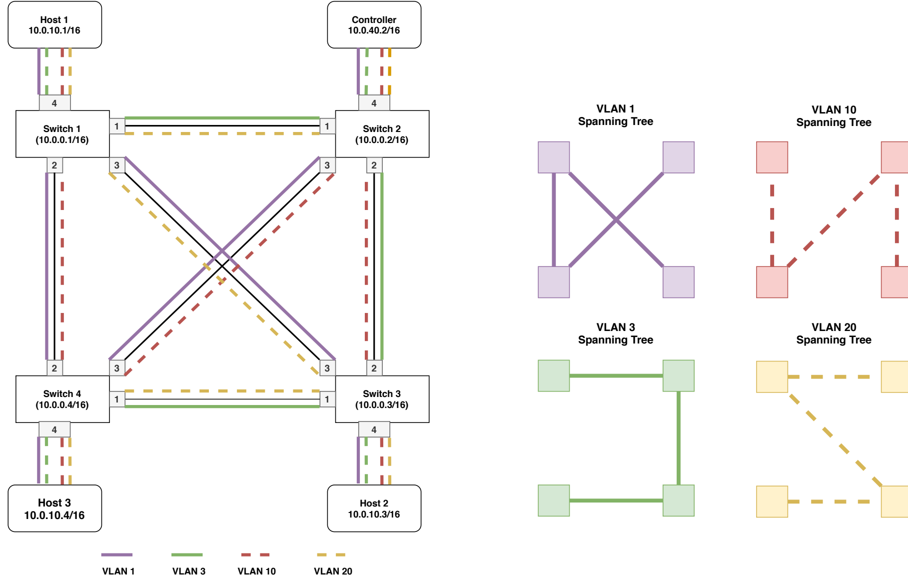

Figure 3: Experiment network topology and VLAN configuration

From our full mesh topology, we generate and test four different spanning trees and assign VLAN IDs to each of them, as shown on the right in Figure 3. As we mentioned in Section 2.4, for an interface to uniquely determine which VLAN a packet belongs to, it must be configured with its native VLAN (into which the untagged traffic will be put). The VLAN spanning tree represented by the solid line is configured to be the native VLAN of the associated interfaces while the VLAN spanning tree represented by the dashed lines is configured to be the trunked VLAN of that interface. For the interfaces that connect with each host, we configure the host to be a member of the default VLAN (VLAN 1) and configure each interface to trunk the rest of the VLANs.

We run the controller on a Ubuntu 16.04 virtual machine with 2 GBytes of RAM and 1 CPU cores running at 2.6 GHz. The endpoints are Windows 10 virtual machines with 4 GBytes of RAM and 2 CPU cores running at 2.21 GHz. The VM is deployed on a windows host machine with 32 GB RAM and a 6 core 2.21GHz CPU. The host that runs endpoints VM is installed with multiple network interface cards that allow each VM to bridged to a separated NIC that connected to different physical switches.

To evaluate the performance overhead of our host agent, we enable different functionality to be performed by each components and measure the cost of each operation separately. Table 1 shows the

Table 1: Host-Based SDN Components and Functionality

	Header Field Modification (Software)	Header Field Modification (Hardware)	Packet Elevation	Flow Table Lookup
Kernel Network Driver	✓	✗	✗	✓
Userspace Service	✗	✗	✓	✗
Network Interface Card	✗	✓	✗	✗

all the essential operations performed by each component.

3.6.2 Packet Header Modifications

We evaluate the host agent's ability to modify the packet based on the controller's instructions. As mentioned in Section 3.5, some header modifications, such as the VLAN tagging and checksum recalculations, can be offloaded to the NIC. In such cases, the kernel driver only needs to specify the offload flag, and the NIC will fulfill the corresponding calculation and insertion request. For other header field modifications, the kernel driver must modify the packet header in software. We will evaluate the hardware-based and software-based header rewriting separately.

To evaluate both types of header rewriting, we use a TCP socket program to generate network flows with different packet generation rates. To determine the overhead of packet modifications, we measure the end-to-end round trip time (RTT) with and without modification under the same forwarding path. In both types of experiments, we exclude the time required to elevate the packet to the controller by locally setting a rule to modify the field to a pre-determined value.

To evaluate packet modifications by the driver, which include header fields from the network or transport layer, we explore the Differentiated Services Code Point (DSCP) field in IPv4 header as a representative example. In this scenario, we only perform the header rewriting on the responding machine. We use Wireshark on the sender to record the time that elapses between the first packet transmission and the receipt of the response from the destination. Such time period captures exactly one packet header modification.

We first show results of the baseline (no modification) case in Figure 4. Figure 5 and Figure 6 show the result of different types of header rewrite. Both type of packet modification, the delay caused by the operation is minimal. The difference between the baseline and the DSCP rewriting appears only in the 64Mbps case and grows to around 1ms in about 50% of cases. In essence, even in software, the driver is able to keep pace with the packet rate.

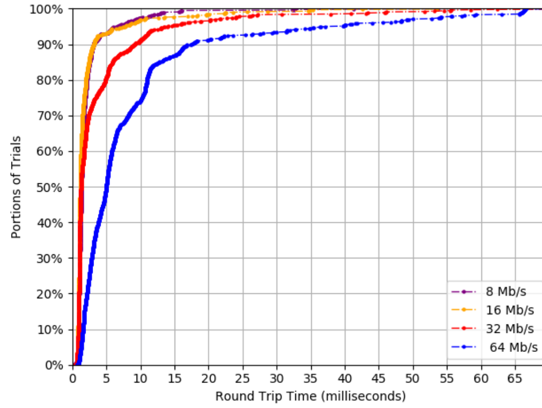

Figure 4: No packet modification

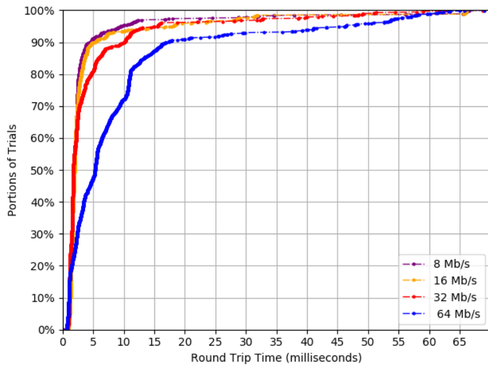

Figure 5: VLAN tagging

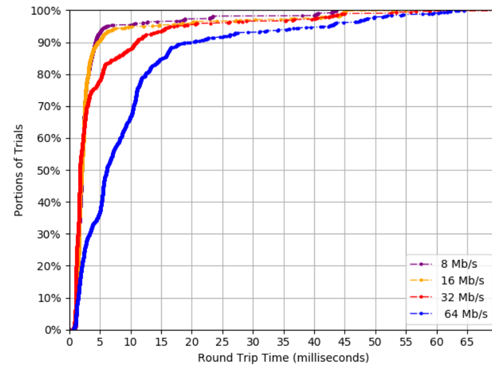

Figure 6: DSCP field modification

When comparing the result between no modification (Figure 4) and VLAN tagging (Figure 5), we see that when the packet rate is under 16Mb/s, the baseline case has a slight advantage that 90% of trials completed under 3 ms, compared to 5ms for the VLAN tagging. But as the packet rate increases, the difference between two cases decreases. When the packet rate is 32Mb/s the difference is less than 1ms (around 9ms for baseline and 10ms for VLAN insertion). Finally, when the packet rate becomes 64Mb/s, both cases require around 17ms to complete the round trip for 90% of the trials. Since, in both cases, the NIC has to process the packet, such result proves the work load caused by the regular packet processing is the dominating factor, especially when the packet rate is high. The extra work load induced by the VLAN tagging causes unnoticeable overhead.

3.6.3 Evaluating Arbitrary Forwarding Path Functionality

We examine the host agent's ability to influence the path used to forward a packet. We confirm that a packet traverses a specific path by using a simple repeater (also called an Ethernet hub) that

broadcasts every bit received on an interface port to all other ports. We install a monitoring device on one of the ports, allowing us to passively confirm the packet transmission on each segment.

While conducting this experiment, we also measure the time required to elevate a packet to the SDN controller, since the controller dictates the path each flow will take. For each new flow a host sends, we record the end-to-end RTT, which includes the delay caused by the elevation, the overhead of one VLAN tagging operation, and the propagation delay. We conducted our experiment using the following four scenarios:

- `1-hop-forwarding`: Host 1 communicates with Host 2 using the VLAN 1 spanning tree.
- `2-hop-forwarding`: Host 1 communicates with Host 3 using the VLAN 20 spanning tree.
- `3-hop-forwarding`: Host 1 communicates with Host 2 using the VLAN 10 spanning tree.
- `asymmetric-forwarding`: Host 1 communicates with Host 2 using VLAN 10 (3 hops) for the outbound path and VLAN 20 (1 hop) for the return path.

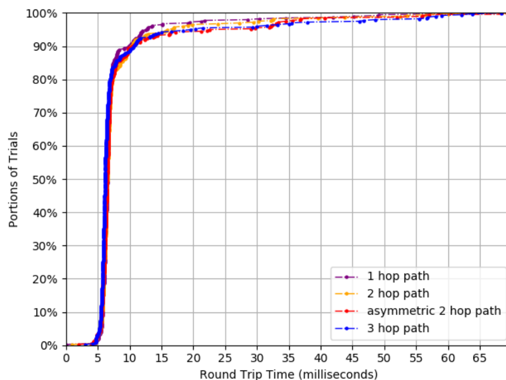

Figure 7: Round trip time with different forwarding path

Comparing Figure 7 to the result of packet modification (Figure 5 and Figure 6) shows that elevation dominate the overall SDN operational costs. When compared with the 8Mb/s case in Figure 5 , by taking the difference, we find that it takes around 3-4ms for a host agent to complete a single elevation. As we will show in our next experiment, this elevation cost is slightly better than an enterprise grade SDN switch. In Figure 7, we also observe that even different cases take different forwarding paths, the propagation delay between them does not differ much. This is because the hop numbers we choose for each case are too small to show the difference.

For the default forwarding case, no elevation or packet tagging is involved, allowing it to serve as the baseline for the propagation delay between two hosts. The other scenarios examine when multiple hops are needed to reach the destination. When conducting the experiments, we found that the prescribed

path indeed followed, that bi-directional communication functioned, and that no TCP re-transmissions or packet loss were observed. We found that dynamically selecting the forwarding path on a per-flow basis does not have side effects in terms of packet loss.

3.6.4 Impact on Host Flow Table Size

As we mentioned in Section 2, for a switch-based SDN, fine-grained flow rules that involve matching filed more than just VLAN and MAC addresses need to be stored in TCAM [50]. As shown by Table 2 below, most of the SDN switches have a flow table that can store less than 5K entries.

Table 2: SDN switch TCAM table comparison

Switch Model	TCAM Table Size (entries)	MAC Table Size (entries)	Data Source
Dell 8132F	750	128k	[36, 16]
HP 5406zl	1500	64k	[36, 27]
Pica8 P-3290	2000	32k	[36, 1]
HP 3800 series	4000	64k	[28]
Cisco Nexus 9000	5000	92k	[12]

For our host-based SDN, we implemented a flow table that was maintained by our network driver. This experiment examines the performance impact as the flow table size grows. To populate entries in the flow table, we first generate flows each using a different source port, each flow will be approved by the controller and associated with a `SetVLANID` action. After the flow table has been populated, we randomly select already-approved flows and reuse them to send traffic between two hosts that are installed with our agent. In this experiment, we enable only the flow look-up and the VLAN tagging operation in the receiver's machine. Using the same metric as the packet modification experiment, we measure the round trip time on the sender's machine. For each flow table size, we sample 30 conversations and plot the graph.

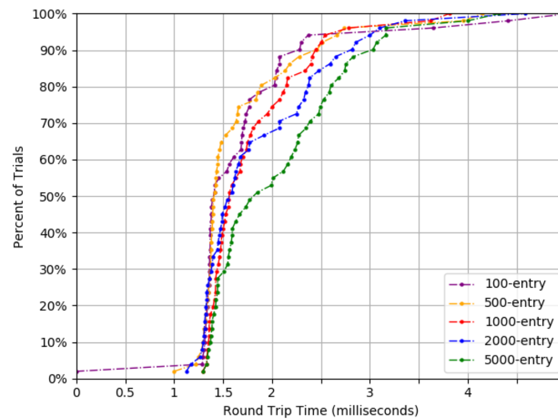

Figure 8: Round trip time under different flow table size

Figure 8 shows that as flow table size increase, the round trip time increase as well which means it takes more time to look up the table and find corresponding actions. When comparing the result between `100-entry` and `5000-entry`, we find that around 40% of the flows take more than 1 msec to complete the round trip. This is due the linked list data structure we are using, which only support entry look-up in linear time. A switch to B^+ tree or hash table can support better look-up time.

3.6.5 Elevation comparison with switch-based SDN agent

We compare the overhead of packet elevation between host agent and switch agent. In this experiment, we use an HP 2920-24G enterprise switch to connect the hosts and the SDN controller. In the switch agent case, we configure the switch with SDN functionality enabled. In the host agent case, we configure the switch as a simple layer 2 learning switch. For the SDN controller, we configured it to involve minimal computation by simply approving every new flow it receives without any parsing and decision making. Figure 9 shows the result of the experiment, the result is taken from our previous work [37].

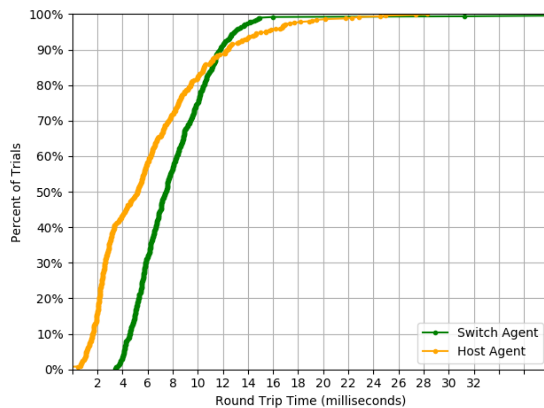

Figure 9: Round trip time comparison between host agent and switch agent

From Figure 9, we can see that the host agent is slightly faster than the switch agents in terms of packet elevation. Both agents requires around 12ms to complete two elevations plus the packet round trip. In essence, the switch-based agents may be using slower generic processing hardware for elevations than is available on endpoints to maintain a reasonable manufacture cost. It is likely that host-based agents will remain at least similar to switch-based SDNs in their elevation performance.

3.6.6 Impact on Legacy Switch Table Size

From Table 2, we can see that traditional managed switches typically have a switch table containing entries ranging from 32k up 128k. For some switches, the number of entries a MAC table contains can

be up to 100 times larger than the number of entries that stored in its TCAM. When a per-VLAN spanning tree is used in a network, it can cause a single MAC address to be associated with multiple VLAN ID, inflating the necessary number of entries. However, this inflation would only occur when the SDN controller wants a host to use a non-default path. Since it can run arbitrary software, the controller is free to optimize its orders to achieve its traffic engineering goals while minimizing table inflation.

In this experiment, we analytically show how the number of entries in a switch's MAC table are affected by the multiple spanning trees. We first define some notations and assumptions for the analysis. Suppose originally in a network that configured with a single spanning tree, a switch has to store a total number of m entries in its MAC table and each entry have a aging time of t seconds. For simplicity, we further assume there are a total number of m' unique MAC addresses among that m entries and each address belongs to a different host. Let's consider a busy network where each of the m' hosts initialize a number of t flows to the other $m' - 1$ hosts during a time period of t.

Given the above setting, let's consider the same network and switch when multiple spanning trees are configured. Suppose our SDN controller only choose to use a non-default spanning tree for p percent of the flows from every hosts and the spanning tree selected for a flow is randomly pooled from a total number of q spanning trees. Under this assumption, each of the m' hosts will add extra entries to the MAC table. The expectation of the number of unique spanning trees used for the fp flows originated from a single host are $q(1 - (1 - \frac{1}{q})^{fp})$. Therefore during a time period of t, the estimated number of entries increased on that switch is $q(1 - (1 - \frac{1}{q})^{fp}) \times m'$.

Given the above analysis, let's consider a large and busy network where there are 10,000 unique addresses in the MAC table of a switch. each of those 10,000 host generates 15 flows per second that destined to the other 10,000 hosts. The SDN controller selects 5% of the flows to use a non-default spanning tree from a total number of 10 spanning trees. As a result, we get that the switch requires a MAC table of 110k entry size to store all the entries for each of the 10,000 hosts. Even under such intense activity, a switch that has 128k MAC table size can also handle this situation.

To address the table inflation, a short aging timer can be set to prune unused entries. For a traditional managed switch, a frame that does not match any existing (MAC, VLAN) pair entry results in the switch broadcasting the packet out each interface. In our approach, each host is a member of VLAN 1 by default. Therefore, when a host is asked to use a non-default VLAN ID to tag its outbound packets, the first packet will be broadcast by the switch. Such broadcasts can be avoided by tagging the ARP packet which is already broadcast. In our implementation, we allow such broadcasts to occur for simplicity.

3.7 Discussion and Conclusion

For operations that are traditionally performed on routers such as NAT or PAT, endpoints can also perform those operations with appropriate extension implementation. But these operations are already supported by legacy routers in modern networks and may not need to be integrated into the SDN. From a practical standpoint, if SDN support for these features were needed, it may be more cost effective to upgrade a small number of routers to support coarse-grained SDN rules than to upgrade all the switches in an enterprise.

Our analysis has examined the ability of a host-based SDN to perform the same forwarding path control and inspection capabilities of a switch-based SDN. The evaluation in Section 3.6 showed that the performance overhead for our system is negligible. Our host-based SDN can store flow rules much like a switch-based SDN, but only for a single hosts. This means the SDN controller is no longer constrained by the limited TCAM table size and can choose to push fine-grained rules even in a large network. More importantly, all the SDN functionality is achieved without using any SDN middleboxes. From a economic stand point, our approach allow user to benefit from the SDN without upgrading any network devices. Such cost reduction may allow organizations to begin mass deployments of SDN technologies even in large enterprise networks.

4 Detecting Host SDN Agent Compromise

We have discussed the benefit of moving SDN agent into the host. Such a shift mainly addresses the visibility and scalability problem for the original switch-based SDN. But such shift also exposes the SDN agent to new risks. Unlike network-based monitoring systems, which runs on an endpoint or middlebox that the network operator fully controls, such host-based SDN agents are running on a user's endpoint. For an adversary who has compromised the user's endpoint, the host agent's reporting can be silenced or even subverted to provide false information. The compromise can be achieved by targeting directly to the host-based SDN agent or through vulnerabilities in other software.

To address such problems, we observe that in some cases, the data provided by a host SDN agent can be corroborated with data from another agent in a remote endpoint. An agent's reports about network flows, for example, can be easily corroborated by examining the data flows reported by agents on other machines. There are also cases where only a portion of the data can be corroborated. For example, a remote host SDN agent may not be able to definitively confirm application layer headers. With only partial verification, it may be possible to determine whether the unverified portion is consistent with the verified data.

In this section, we focus on the host-based SDN agent and examine how its data reporting via the OpenFlow protocol can be evaluated using data from other SDN nodes. For an SDN agent, it first seeks guidance from the controller whenever a flow that lacks a matching entry in the local rule cache is encountered. When multiple SDN agents exist along the communication path, multiple independent reports are receive on the SDN controller. If the controller detects any inconsistency from those reports, then it may be able to pinpoint a compromised host agent.

This problem is not unique to our host-based SDN agent. Other security systems, such as firewall or host-based intrusion detection system that adopt the similar reporting mechanism as the SDN agent face the same problem. Therefore, the approach can be adopted by other endpoint security systems.

4.1 Corroborated Host-Based SDN Agent Enforcement

In this section, we provide example attacks, their consequences, and how corroborated sensing can help. We then describe the system and threat model we are considering. We then describe the Corroborated Host-Based OpenFlow Sensor Enforcement (CHOSE) system and scenarios in which it is effective.

4.1.1 System Overview and Threat Model

In Figure 10, we provide an example local area network for an organization. The organization's network is connected to the Internet via a gateway router. The network has a set of switches, each of which are legacy switches. An SDN controller manages the host agents that are installed on the

hosts. Communication in this network may be external, such as Host 1 communicating to a host on the Internet, or internal, such as Host 1 communicating to Host 2.

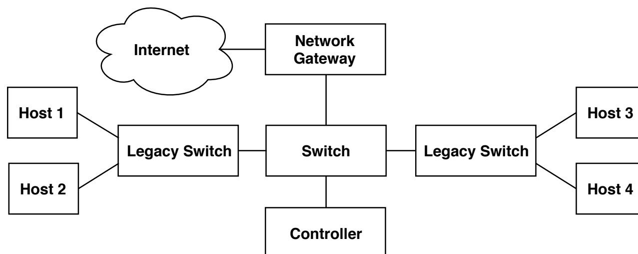

Figure 10: An example enterprise network with SDN agents on each end-point.

In this example, the trusted computing base (TCB) includes the SDN controller and the network switches. The physical connections between the switches, hosts, and controllers are considered uncompromised and reliable.

The trusted computing base does not include the host. The report from the SDN agents could be erroneous or absent due to attacks that have compromise the kernel components that the SDN agent depend on. In some cases, a set of compromised host agents may collude to with a goal of evading detection. There are also cases where a host with a compromised agent tries to communicate with another host with a properly working agent. In both cases, as long as there is an uncompromised SDN agent along the communication path, the controller can detect the inconsistency from their flow reports. Such uncompromised SDN agents can be an host agent or a switch agent. If it is a switch agent, we consider it to be the TCB. We configure such switch agent to elevate the first packet of every new flow it receives so the SDN controller can receive a complete report.

We consider an adversary who focuses on maintaining persistence, the ability to move laterally within an organization, and to maintain communication with a command and control system. That adversary requires covert communication channels. Such an adversary would forgo resource exhaustion DoS attacks since they are easily detected and can be trivially mitigated by prior work [7]. Accordingly, we omit any further analysis of DoS attacks.

The defender's goal is to receive a full reporting of all communication flows that occur in the network in a logically-centralized controller. The defender wants to block any flow requests from sensors that are inconsistent with other sensors. With this full accounting of flows, the defender can construct arbitrary access control policies on the controller. Since the development of effective network access control policy is its own active research area, we consider it beyond the scope of this work.

4.1.2 Corroborated Sensing Deployment Scenarios

When hosts are deployed with SDN agents, each will act as an independent reporting system. When network communications happens between such hosts, the SDN agents are essentially report flow information about the same communication.

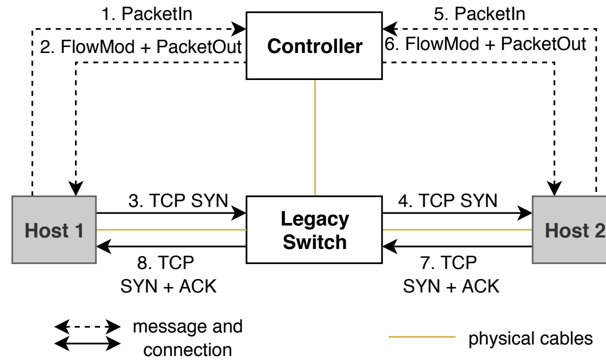

Figure 11: When both endpoints run an SDN agent, if either is uncompromised, that uncompromised agent will alert the central coordinator of inconsistencies via its `PacketIn` data.

Using the network in Figure 11 as an example, consider Host 1 and Host 2 are communicating through a TCP connection. As shown by the graph, the same TCP `SYN` packet have been reported to the controller twice through the `PacketIn` as indicated by line 1 from Host 1 and line 5 from Host 2. In this case, if the SDN agent on either Host 1 or Host 2 provide faulty flow information about the `SYN` packet, or refuse to engage in the packet elevation process, the controller is able to detect such mismatch.

This detection mechanism goes to the heart of the attacker's goals. To establish communication for command and control or to propagate the attack to other machines, the adversary must establish new connections. However, an endpoint with SDN agent will reveal this flow when the adversary makes the connection attempt, causing the adversary to be detected. The attacker must alter a sensor to avoid this reporting, but any alteration will result in a mismatch on the remote host's agent.

In the Figure 11 example, the controller will receive conflicting information and know one of the two hosts is compromised, but will not know which SDN agent is the one that provide faulty information. Such information is helpful in terms of containing the attack, but to definitely determine the compromised SDN agent, it require other report source. If in the communication path, the switch is an SDN enabled switch as shown in Figure 12, the controller can determine which host is deceptive. The SDN enabled switch would provide information about the `SYN` packet (shown by line 4). Further, since the SDN switch is in the network's TCB, its reports can be used as ground-truth data. Without a ground-truth, network operators would need to check both hosts for a potential compromise.

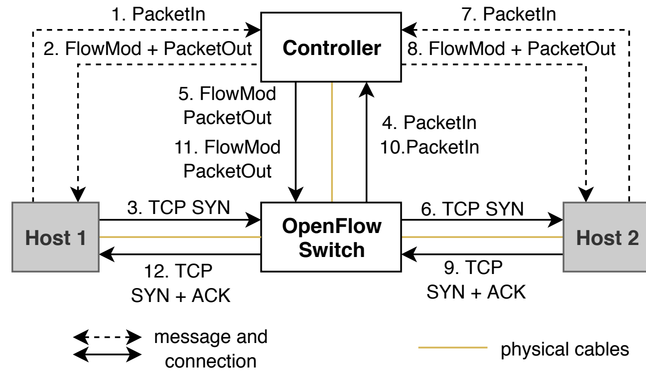

Figure 12: When an SDN switch is on the network path, the controller receives `PacketIn` data that allows it to identify which endpoint, if any, is faulty.

When two host-based SDN agent corroborate to detect agent compromises, the controller must be careful with what rules it pushes to the agents. As showed in figure 13, when there are only SDN agent running in the hosts, the controller needs to use uni-directional `FlowMod` to detect if either of the host agent elevate the faulty flow information.

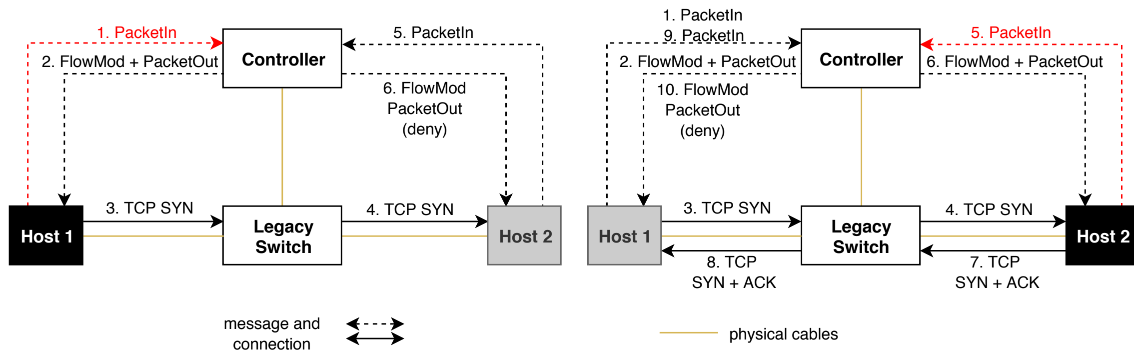

Figure 13: When one of the hosts is compromised (shaded in black), the controller will notice a discrepancy when receiving a `PacketIn` from the non-compromised host (shaded in gray).

In Figure 13, we show the process that would occur if either Host 1 or Host 2 was compromised in this example scenario. In the event Host 1 is compromised (the left diagram in Figure 13), it could fail to send a `PacketIn` in Step 1 or send an inaccurate `PacketIn` (e.g., a `PacketIn` with inaccurate payload or header information) and receive the controller's approval. However, Host 2 would then send a `PacketIn` in Step 5 and the controller would notice the discrepancy between the two `PacketIn` messages, deny the flow in Step 6 and drop all the packets in the flow, preventing the application at Host 2 from receiving them.

If Host 2 were compromised, which is depicted in the right side of Figure 13, a similar process would occur, but the detection would be slightly delayed. In this case, the first 4 steps would proceed and Host 2 would either neglect to provide a `PacketIn` in Step 5 or provide inaccurate information.

Since the SYN packet would already have reached Host 2 in Step 4, Host 2 could process the message and respond in Step 7. However, if the controller only pushes a unidirectional FlowMod rule in Step 2, Host 1 would again elevate the packet to the controller in Step 9. At that point, the controller would note that Host 2 failed to send a proper PacketIn associated with the SYN packet and would insert a denial FlowMod into Host 1 in Step 10, preventing the application at Host 1 from communicating with the compromised host.

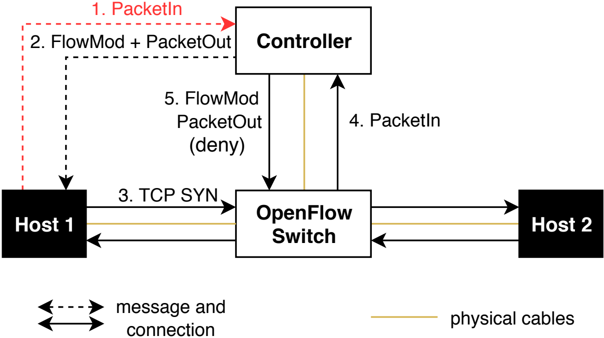

Figure 14: If both hosts collude (shaded in black), only a middlebox or an SDN switch between the hosts can be used to detect the malicious flows.

When both hosts are malicious and only legacy switches connect the hosts, it is possible for the hosts to collude and choose not to elevate packets to the controller. Without a middlebox or an SDN switch that connects the devices (as shown in Figure 14), this scenario cannot be avoided.

4.2 Uncorroborated Data in Host SDN Agent

For the host-based SDN agent, the flow information it can provide through PacketIn can contain more than just the packet itself. Because of the visibility into the host. The PacketIn can also contain information about the user and originating application. Such contextual information is only available at the host that originate the packet therefore cannot be easily matched like the port and address information.

There are cases a controller may be able to detect obvious signs of forgery, such as a connection on port 22, commonly associated with the SSH protocol, purportedly originating from an email client. In this case, the controller needs to know what applications are network flows usually associated with. When an trusted agent (such as an SDN switch) reports authentic flow information, the controller can use this information to look up its supposed application and compare it with the next or previous report that describe the same flow. An application discrepancy indicate the sign of agent compromise. However, for a sophisticated attacker, he would likely be able to craft contextual data that would plausibly be associated with the verifiable network headers and packet payload.

To create an association between the network flows and their originating applications, learning-based algorithm might play a role. Previous work [3] has shown that by observing a sequence of packets of a network flow, a learning-based algorithm can classify such a packet sequence to its originating application. Such an approach does not depend on the payload of the network packet but only use header information and other collected statistics. This approach, although provide extra application information, cannot give affirmative answers due to the accuracy of packet classification algorithms [49]. Also, the flow report from a host SDN agent may contains information like process identifier that is verifiable only on the host itself. To gain trust in information that cannot be corroborated or inferred, trusted hardware may play a role. However, when a root of trust or a trusted execution environment is established, one must ensure that the dependent components are all in the trust chain or all in the trusted environment. Such a requirement might place extra as constraint that limits the appliability of these approaches.

4.3 Implementing the CHOSE System

The CHOSE system has three components: 1) a switch-based SDN agent on physical switch that comply with the OpenFlow protocol, 2) the host-based SDN agent as we described in Section 3, and 3) a custom SDN controller that manages connections for both switch and host SDN agents. The implementation of the host agent have already been discussed in Section 3 and the switch agent is running on a enterprise grade SDN switch. In this section, we focus on the implementation of the SDN controller.

4.3.1 SDN Controller Customization

The SDN controller must support both the host-based SDN agent and communication from traditional agents running on switches. This controller distinguishes the different SDN agent type based on the destination transport layer port and handles the communication in separate threads of execution.

When receiving a `PacketIn`, the controller must determine what SDN agents would be on the path from the source machine to the destination for that flow. If the `PacketIn` arrives from the first expected SDN agent on the path, the controller consults its normal policy rules to determine whether the flow should be allowed. If not, it sends `FlowMod` and `PacketOut` messages to the agent that order the packet and all other packets in the flow to be dropped. If the controller policy dictates the flow should be allowed, the controller stores a record of the flow in a local list of active flows and then sends `FlowMod` and `PacketOut` messages to the requesting SDN agent to approve the source to destination direction of the flow.

Since the controller sent a `FlowMod` only to the originating SDN agent during its approval, subse-

quent SDN agents on the path will again elevate the packet to the controller. If the controller receives a `PacketIn` from an SDN agent, and that agent is not the first agent that should have appeared on the flow, the controller will check to see if it already has an entry for the flow in its active flows list. If it does not, the controller will send `FlowMod` and `PacketOut` messages to the agent that order the packet and all other packets in the flow to be dropped. It will also make note of the SDN agent that failed to elevate the flow. Alternatively, if the controller sees that the flow is in its active list and was previously approved, it will order the SDN agent to approve the flow.

In this approach, the controller makes only unidirectional forwarding approvals in its `FlowMod` messages. This is essential to detecting compromised or malfunctioning agents that are at or near the destination. When a reply is issued, such as the `SYN+ACK` packet in a TCP connection, each agent on the reverse path will again elevate the packet to the controller. At that point, the controller can confirm it has received all the expected requests from agents in the original direction. It can then send a `FlowMod` message that updates the original uni-directional flow approval to instead allow bi-directional communication on each agent on the path.

With this approach, the controller receives corroboration on packets elevated from each OpenFlow agent any time there are multiple OpenFlow agents on the path. Further, if at least one OpenFlow agent on the path is not compromised, the controller will be able to detect the existence of any compromised agent on the path that omitted or modified the flow information.

4.4 Evaluating the Security and Performance of CHOSE

In this section, we describe our experimental setup, our performance evaluation process and results, and the security evaluation methodology and results. In our evaluation, we aim to answer two questions: 1) What overhead does corroborated sensing introduce to an existing SDN deployment? 2) What security guarantees does corroborated sensing offer to such a system?

4.4.1 Experiment Setup

In both our performance and security evaluation, we configure our network to match Figure 11. We use an HP 2920-24G enterprise switch with its SDN functionality enabled to connect our hosts and controller. Our controller runs on a laptop that runs VirtualBox to host Ubuntu 16.04 VM. We configure the controller with 2 IP address and place one of the IP addresses under the control of SDN so that the communication between sensors and controller will also be subject to the SDN's elevation model. We then connect two end-hosts to the switch. The first host is a Mac mini that runs VirtualBox to host a Windows 10 VM. The second host is a Macbook Pro that runs VirtualBox to host a Windows 10 VM.

4.4.2 Performance Evaluation

To determine the overhead associated with corroborated sensing, we compare it with regular SDN behavior in both switch-based and host-based SDN configurations. We create a HTTP client program using `winsock2` to connect to HTTP server written with the `Mongoose` web server library. Our client creates connections in a serial fashion.

We ensure that the tested SDN agents on the hosts and switch will perform a flow elevation for each new connection. Since the system uses `FlowMod` rules to avoid elevating subsequent packets in a flow, the overheads associated with packet elevations will only affect the first round trip in a flow. Accordingly, we use the round trip time (RTT) on the first set of packets in the flow (e.g., the `SYN` and `SYN+ACK` TCP packets) as our performance metric.

By comparing the time required under varying deployment scenarios, we can determine the latency associated with elevation requests from switch and end-host agents along with the time required for the controller to correlate flow requests. We use the following four scenarios in our testing:

- **Scenario 1: Switch-Based SDN Agent Only:** In this scenario, neither of the hosts run an SDN agent and simply transmit packets using the native Windows networking stack. The physical SDN switch elevates each new connection it sees to the SDN controller. The controller only processes standard OpenFlow packets and it approves all new flow requests it receives. Since this controller engages in minimal computation, this scenario provides a baseline for a physical switch's performance.

- **Scenario 2: Host-Based SDN Agent Only:** In this scenario, both of the hosts run our Windows SDN agents. The agents gather end-host application context and flow data and include this information in elevation requests to the controller for each new connection. In this case, the controller processes the modified OpenFlow messages for the host sensors. However, the physical SDN switch is configured as a simple learning switch and does not send any elevation requests to the controller.

- **Scenario 3: Both Switch and Host Agent:** This scenario uses OpenFlow agents at both hosts and the physical switch. The controller processes packet elevations from both types of agents. However, the controller statelessly approves the flows independently and does not perform any correlation or analysis of the requests across SDN agents.

- **Scenario 4: Full Sensing and Flow Correlation:** In this scenario, the SDN agents run at the hosts and the physical switch. The controller examines the elevations across SDN agents and correlates the requests to identify discrepancies or missing elevation requests.

4.4.3 Round Trip Timings

For each of these scenarios, we conduct 500 trials, with each trial consisting of a new connection in which the RTT for the initial packets are measured. Once the connection is established, it is immediately terminated and the next trial begins. We present the results of these trials in Figure 15.

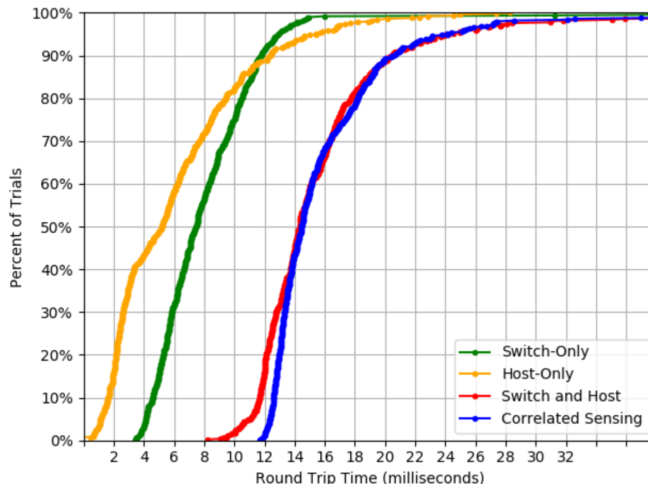

Figure 15: Round-trip time of serial connections across 500 trials.

The overhead of corroborated sensing is the timing difference between the third scenario, in which host and switch agents are used but the controller acts statelessly, and the fourth scenario, in which the SDN agents are identical as in the third scenario but the controller correlates flows across SDN agents. In Figure 15, the distribution curve of the round trip times associated with these two scenarios largely overlap, indicating that the performance costs of corroborated sensing are not significant.

From these experiments, we see that most flows complete in less than 15 milliseconds, even with corroborated sensing, and that around 90% of flows complete in less than 20 milliseconds. The performance of the host-based only sensor is faster in most cases than the switch-only sensor. This appears to be due to the physical switch using its relatively-slow integrated processor for performing flow elevations and the TCAM that store the flow is relatively slow to access. As one might expect, the RTTs in the third scenario, which requires elevations from the hosts and the switch, are roughly the sum of the times in Scenarios 1 and 2.

4.5 Security Evaluation

We examine the effectiveness of the corroborated sensing approach using the configuration described by Scenario 4 in the performance evaluation. We create four cases in which we vary the proper operation status of the client and the server. Across the four possible combinations, we vary whether

the host elevates packets normally or whether it evades proper operation by not elevating the packet appropriately.

Table 3: Number of connections allowed and denied by scenario.

Case Number	Client Status	Server Status	Client Flows Approved	Server Flows Approved	Client Flows Rejected	Server Flows Rejected
1	Normal	Normal	500	500	0	0
2	Normal	Evades	500	0	0	500
3	Evades	Normal	0	N/A	500	N/A
4	Evades	Evades	0	N/A	500	N/A

In Table 3, we show the results of testing these four cases across 500 trials each. As expected, when both the client and server are operating normally, all the flows are approved. In the second case, where the client acts properly but the server agent does not, the initial packets are approved and reach the server, but the server's responses are dropped because the server failed to elevate both the client's original packet and the server's response packet to the controller. Scenarios 3 and 4 proceed identically since the controller denies the packets when the SDN switch elevates them because the client failed to originally elevate the packets. In that case, the packets are discarded before the server can receive them, so the server never knows to create a response.

As we discussed in Section 4.1.2, if the switch between the hosts is legacy, the uncompromised host triggers the controller's detection rather than the SDN switch. Further, if both hosts are compromised with a legacy switch, the communication goes undetected. We omit these cases for brevity.

In these experiments, we simply disable the sensor rather than having it create forged data. Since the flow decisions use the network tuple (IP addresses, ports, and transport protocol), any alteration of these fields would constitute a new flow and thus the forgery in an elevation request would cause the actual packets to not match a flow rule when an uncompromised agent elevates the packet, resulting in a drop rule by the controller. Alterations of other fields in the packet headers could be detected simply by including those fields in the controller's local active flows table.

4.6 Discussion

In this section, we examine how network operators can detect compromises that affect the accuracy of data reported by host-based SDN agent by correlating that data with other sensors in the network. We show that if a single non-compromised sensor exists on the network path a flow takes, a centralized network controller can detect discrepancies in the information reported by any compromised sensors on that same path with perfect accuracy. Our performance results show that this corroborated sensing comes with little extra cost over a standard OpenFlow deployment. In around 90% of cases, the round trip time of the first packet exchange in a connection took less than 20 milliseconds, which includes

all of the required flow elevation. Since this flow elevation occurs only during the first round trip of a new flow, these overheads are unlikely to affect the user experience while offering tangible security benefits.

5 Monitoring on Server Endpoint

In this section we move our focus to software on the server endpoint. Unlike client-side software, software on the server endpoint, such as a web server or mail server, usually has to provide service for multiple users at the same time. The resources and execution context on that single server is being shared among different users. Such sharing can cause security problems on both the server side and client side. In many cases, the server is acting as fullly privileged intermediate proxy between the users and back-end resources. This situation can allow adversaries to modify or alter other user's back-end data in cases of a server compromise. An adversary may further seek chances to propagate the threat to other users when they access the same compromised server. Server application's action are usually hard to analyze as well. Simply monitoring the execution trace of the server application will only generate a sequence of intertwined data. In the case of an attack, the defender cannot easily attribute execution traces to different user's requests in order to locate the source.

5.1 Single-Use Server Model

To address the problem mentioned above, we explore an one-to-one client-server (C/S) model that isolates different users even when they are accessing the server at the same time. The one-to-one C/S model can gives each user a separated execution context that allows us to exercise a least privilege strategy. With such a separation, each server instance can be constrained differently based on the user's role. This separation helps with the monitoring as well, since each server instance only generates the execution trace of a single user. This naturally classifies the monitoring data into different groups and narrows down the scope if any of those data require further analysis. Previous work has tried building such server model using virtual machines [52], but VMs are heavy-weighted and resource-consuming: That approach cannot scale to a large number of server instance.

In this section, we give an overview of a single-use server model that we design and implement. Then, we focus on the monitoring infrastructures we build to work with the single-user server model. Instead of using virtual machines, our single-use server model leverages containerization as a light-weight runtime to hold multiple server instances. Container also benefit the monitoring. As mentioned in Section 2.7, unlike virtual machine, containers share the same system with the host but still provide a certain level of isolation. This allow us to clearly view any in-container activity without using hypervisor monitor approach. Our single-use server model consists of the following components:

- **Application Container:** The container that runs the server applications. In our design, we consider the application container as untrusted and can be vulnerable to attacks.
- **Container Manager:** This entity is responsible for allocating and recycling of any application

containers. It manages a pool of running application containers. The manager will respond the client-to-SuS middlebox when it receive a request that ask for new containers. The manager also needs to maintain back-end account and password information that associated with each application container.

- **Authentication Container:** For applications that request user to authenticate, this container retrieve the credential and perform any necessary validation on behalf of the application container. The authentication container also communicate with the container manager in case where a permission adjustment is required.
- **Client-to-SuS Middlebox:** This middlebox act as a request demultiplexer for the SuS model. Any client data must reach this middlebox first before being forwarded to their assigned server instance. This middlebox also perform monitoring, logging of user's network trace.
- **SuS-to-Backend Middlebox:** This middlebox perform monitoring and logging of communications between the application container and back-end resources. When retrieving data from the back-end resource, it will compare the command with a set of approved actions to detect any unapproved access.

In our single-use server model, a server application is placed in the application containers. Each application container is initially configured with a least-privileged back-end resource account. When a user access the server, the client-to-SuS middlebox will determine which application container this request should be forwarded to. When a user authenticate, the authentication container will verify user's crendential and inform the manager to adjust the permission to match the priviledge associated with the authenticated user. When a user exit or remain inactive for a period of time, the client-to-SuS middlebox will inform the container manager to reclaim this inactive container.

5.2 Client-to-SuS Middlebox

One problem with running such a single-use server model is that we have to differentiate different users and assign them to the correct server instance. For different server applications, the approach might differ. For a SSH server, a user needs to authenticate itself before the access is granted, in such case, its original authentication process can be incoparate into the autentication container to build portal that first identify the user and then dispatch the user to its own server instance. If a user is allowed to access the server first as an anonymous user and then authenticate itself, the process will be a lot more complicated. In this section, we use web server as an example, and discuss the design and implementation how the client-to-SuS middlebox can differentiate different user and proxy their request to the assigned web container.

In a web application, a user's requests are usually in a HTTPS packets, our font-end middlebox first perform a SSL termination that not only alleviate server instance of the extra computational pressure but also reveal the HTTP header that allow us to use customized header filed to differentiate users before performing any forwarding. To identify a user, our approach uses an HTTP cookie, which we call the `SUS_DEMULTIPLEX_COOKIE`. If a user contacts with the client-to-SuS middlebox without or with an expired cookie, the middlebox treats it as a new client and asks the container manager to assign a new container for this user. After receiving the response, the middlebox forwards the user to its newly assigned container. Upon receiving the first response from the server instance, the middlebox uses a `Set-Cookie` header filed to transfer the cookie back to user so that the next request from that user will contain the same cookie information to identify itself. The middlbox needs to updates an internal mapping structure between the cookie value and the container IP address. In cases where a user's request already contain a valid `SUS_DEMULTIPLEX_COOKIE` cookie, the middlebox performs a mapping lookup to retrieve the appropriate container IP and forwards the requests accordingly. In our design, each entry in the mapping structure contains a timestamp value which is also maintained by the middlebox and is used to track inactive containers and allow the container manager to perform any necessary container pruning.

5.3 Container Auditing

The client-to-SuS middlebox discussed allows us to obtain a clean view of the user's requests. Using the information between the user's identity and its assigned container IP, we can easily separate different user's requests. Such separation greatly reduced the number of requests that can reach the server instance. Therefore, the server's response actions are also simplified. If we are able to monitor and record theses actions, a connection between a single user's request and the response of that request can be built to help analyze a server application's action. Since the Docker framework is one of the most popular in the market [15], in this section we focus on the Docker containers and discuss the detail of how we leverage the Linux's auditing system to monitor and record the action of each server application container.

The Linux auditing system is a native feature on Linux that collects different system activities. It works at the kernel level and can observe every process's operation, such as system calls or access to the file system. Although it is a versatile tool for monitoring process, it does not have a view inside each container. When a host machine runs multiple server instance, each within a container, the native auditing system can only observe different audit records from different processes but cannot associate the audit data with its container if the processes that generate the audit record is running within a container. To address this problem, we design and implemented a container auditing program that log

audit record on a per container basis.

5.3.1 Overview

As we mentioned above, the native auditing system only views the process that runs on a host machine. To get a per container audit record, an obvious approach is to install the auditing program inside of each container and log the record along with the container's identity. Then, all the logged record are generated by in-container processes. Such an approach is straightforward but very costly. Because it requires each of the containers to perform extra works, in a host with a large number of running containers, this approach may not scale well. Therefore, we perform the auditing from outside the container and try to determine if a audit record belongs to a in-container process or not.

5.3.2 Container Startup

To achieve the above goal, we must first understand the startup process of a Docker container. In Listing 1, we shows the process hierarchy of a running container.

Listing 1: Process hierarchy of a running container

```
systemd,1 splash
   containerd, 1321
        containerd-shim, 2740 ...
             apache2, 2772 -DFORGROUND
                  apache2, 3193 -DFORGROUND
                  apache2, 3194 -DFORGROUND
                  apache2, 3196 -DFORGROUND
```

`containerd` is a process that manages the container life cycle. When a container starts, the `containerd` needs to invoke `RunC`, which according to the OCI (Open Container Initiative) specification, will load the the image and execute the command that starts the application process. This process makes `Runc` the parent process of the container application process. After `RunC` finishes its job, it will exit and a re-parenting process makes `containerd-shim` the new parent process of application process. After the entire procedure is finished, a container is setup.

5.3.3 Container Auditing Design

Processes that are started during the container startup will have `containerd -shim` as their direct parent. For processes that are started later, we can iterate up through the process tree to find `containerd-shim` in the path. For processes that are not running within the container, the same process tree searching will end at `systemd` without traversing a `containerd-shim` node. With this

knowledge, we can label all the process on a host machine with container information and separate the system-wide audit records and associate them to each running container. To achieve the above goal, we need the following functionality:

Notification of the Process Event: To determine whether a process belongs to a container, a straightforward method require us to run the `pstree` command every time we receive an audit record. Such an approach can be very costly if a considerable number of auditing rules are added. Instead, we decide to maintain a process tree so our container auditing program can directly access this in-memory process tree whenever it is needed. With correct data structures, the program can quickly iterate through the tree and determine if a audit record belongs to a in-container process.

To build an in-memory process tree, we need a real-time notifications of the system's process events. When receiving a process start event, our program needs to insert a new node into the tree and delete a node if a process exit event is received. Except for maintaining the in-memory process tree, process event notifications also allow us to immediately become aware when a container is started. By examining the process name and looking for `RunC` and `containerd-shim`, we gain the parent process's identifier of any in-container processes. Such information allows us to determine whether a process tree iteration should stop or continue.

Notification of Docker Container Event: The functionality mentioned above allow us to quickly determine whether a process is an in-container process or not. Such information is insufficient because we still do not know which container an audit record belongs to. This problem can be solved when we have notifications of Docker container events. When a container is started, we will receive a notification about the container's identifier along with its initial process's identifier. Because the `containerd-shim` process is usually the direct parent of this initial process, we are able to create a mapping between `containerd-shim`'s identifier with container's identifier.

When the above two approaches are combined, we can start performing a top-down labeling of the entire process tree as we receive more process events. As showen in Figure 16, when inserting a node into the process tree, the container for this newly created process can be immediately determined by checking the label information of its direct parents. If its parent process is unlabeled or labeled as a non-container process, then this child process is labeled as a non-container process. Otherwise, it is labeled as a in-container process and we associate this new in-container process with its container's identifier.

Linux Auditing Record The last functionality we need is to receive Linux's auditing record in our program. The Linux's native auditing program consists of two parts: a kernel module named `kauditd` that perform the actual collection of system's auditing record and a userspace service named `auditd` that receive record from the kernel. Since the `auditd` does not consider an in-container process's auditing record differently, we cannot directly use the native userspace program. To get the auditing

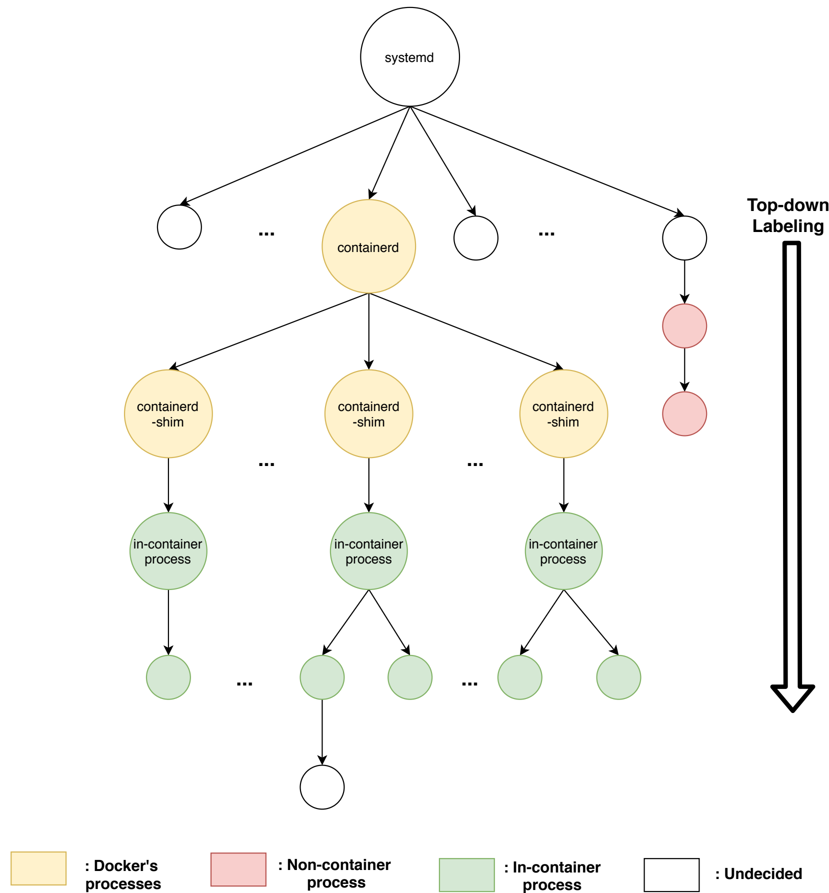

Figure 16: Tree structure to determine if a process is a in-container process

record, we must receive messages from `kauditd`. One way to achieve this is through netlink socket communication which requires a `NETLINK_AUDIT` socket type and parsing of the kernel's message to get the record. Another method is to leverage an audit record dispatcher, `audispd`, which can parse and forward the kernel's message to any user defined `audispd` plugin. We use the latter method to receive our audit record.

To summarize, with notifications of Docker container events and the system's process event, we are able to create a in-memory process tree that helps us determing in-container processes and associates them with their container identifier. With the help of audit record dispatcher, we can receive a complete audit record from the system and perform further processing for each record.

5.3.4 Container Auditing Implementation

We implement our container auditing program using the C++ language. To work with Linux's native auditing system, we use the `libaudit` [44] and `libauparse` [11] libraries. To get notifications of Docker container event, we pipe the output of the shell command `docker event` to our program. To

get process event notification, we implement a netlink socket that receives and parses the kernel's process event notification. For the process tree, we implement it using the map structure from C++'s standard library. Each process node is defined as an entry which can be accessed from its process's identifier. For each process node, we implement a struct that contains its parent process identifier and another map that store the identifiers of children processes. Using this data structure, we are able to quickly locate a process in the tree and perform parent tracing to determine its container information if it is an in-container process.

The overall structure of our program is shown in Figure 17. Each of the components mentioned in Section 5.3.3 update the process tree structure. We implement another queue that allows the `audispd` program to add an in-container process's audit record into the queue and the worker process will perform any further parsing and adding container identity information with each audit record. The reason for such implementation is because the kernel space only has a very limited buffer to store audit record and the `auditspd` tool can only send one record at a time. Therefore, our program must first process each record as fast as possible to not overflow the kernel buffer.

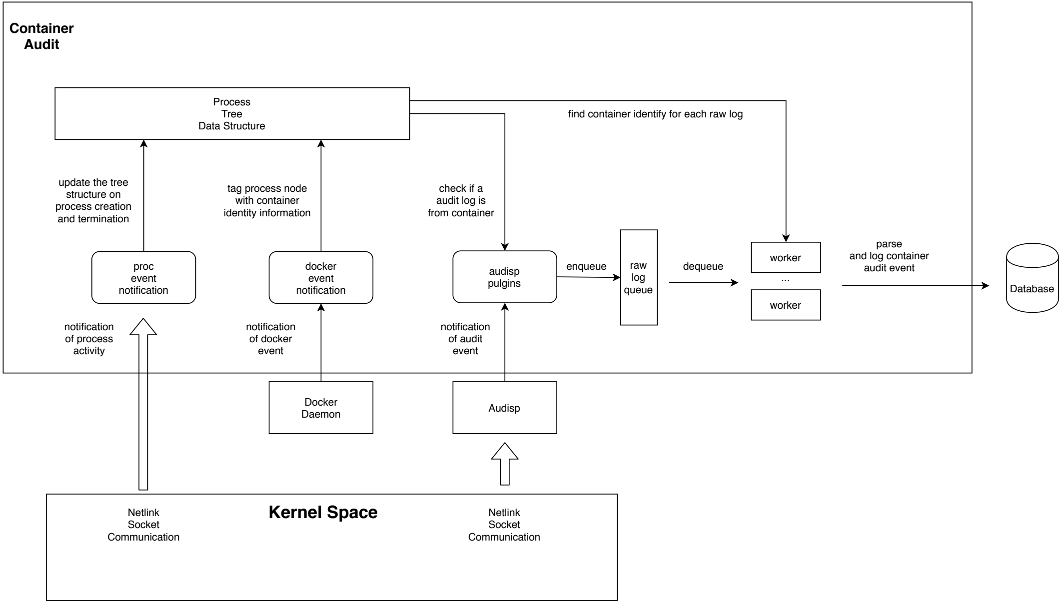

Figure 17: The components in the container audit program

5.3.5 Evaluation

In this section, we evaluate our container audit program by verifying if it can indeed differentiate audit records that belongs to an in-container process and a non-container process.

To conduct the experiment, we use system call audit rules as an example. We first specify three different system call audit rules using the `auditctl` program. Listing 2 shows command to add system

call rules. For each of the rules, we specify two different system calls using the `-S` parameter and name each rule after the `-k` parameter.

Listing 2: System call audit rules specified for the experiment

```
sudo auditctl -a always,exit -F arch=b64 -S getpid -S getppid -k "test1-syscall"
sudo auditctl -a always,exit -F arch=b64 -S mmap -S brk -k "test2-syscall"
sudo auditctl -a always,exit -F arch=b64 -S setitimer -S getitimer -k "test3-syscall"
```

On a host machine with multiple running containers, we write three testing program which we name `test1`, `test2` and `test3`. These three programs will repeatedly execute the system calls specified in the audit rule that ared named after the program's name. Program `test1` and `test2` run inside different containers while program `test3` runs on the host machine. We record the logged audit records from our program and check for any incorrect association between the container's identity and audit record. We found that our container program can correctly differentiate audit rules that are generated by processes that are in different container. Our program never mistakenly records any audit record generated by non-container processes.

5.3.6 Discussion

In this section, we discussed the monitoring component that we implement to work with the single-user server model. Our evaluation showed that the container auditing program can correctly log audit record on a per-container basis. Future evaluations will examine the performance of our container auditing tool, such as evaluating the auditing program's CPU and memory usage as more rules added. We also need to determine what types of audit rules can play a significant role in capturing server application's action.

6 Future Work and Conclusion

The goal of this work is to address some limitations of the current approaches and technologies for network and endpoint monitoring. In Section 3, we focus on the host-based SDN system and note that previous attempts to move SDN agents from switches to host only realize part of the functionality of switch-based SDN. The missing functionality may limit host-based SDN deployment as a versatile security system. Our implementation empowers host-based SDNs with forwarding path control and packet rewriting while keeping the SDN's original design paradigm. Such improvements give host-based SDNs more control in the network that other network-based monitoring, but also grant them with visibility into the software's action like a host-based monitoring approach. In Section 4, we focus on the endpoint compromise problem that is faced by our host-based SDN. We made the observations that multiple host agents in a network are a set of independent reporting systems, that in many cases, report about the same endpoint activity. Such information redundancy can be used to help detect agent compromises. Based on this observation, we implement an SDN controller that corroborate individual agents' reporting to detect any faulty or inaccurate flow reports which indicate a sign of agent compromise. We also realize that such detection mechanisms and deployment of multiple agent in a network is essential to form defense-in-depth in which each agent's control overlaps with other. Such security benefit might have other potential applications we might explore in the future. In Section 5, we addressed the difficulty of monitoring server application's action. By taking advantage of a single-use server model, we implement monitoring infrastructures that first separate different users' requests at the front-end and log audit records that describe each server instance's response to that single user. The separation between different users allow us to get a clean view of user's request and server's response. Our infrastructure greatly reduced the monitoring scope. The audit records when associated to the network traffic build a connection between the consequence and cause, allowing us to understand the way a server application works at functional level. Taking advantage of container's disposable feature, these data record can even help us replay the past event step by step. Such replay functionality will help us to ease the forensic analysis. We hope to build a tool that help defenders to pinpoint the cause of the problem. In the future, we will explore to what extent we can facilitate the forensic analysis process by leveraging the separation under the single-user server model.

References

[1] P. 8. Pica 8 open networking. http://www.tooyum.com/download/pica8-datasheet-48x1gbe-p3290-p3295.pdf. Accessed: 2020-04-29.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[3] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescape. Traffic classification of mobile apps through multi-classification. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, 2017.

[4] M. Ali Babar and B. Ramsey. Understanding container isolation mechanisms for building security-sensitive private cloud, 01 2017.

[5] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth. Evading machine learning malware detection. *black Hat*, 2017.

[6] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. Hima: A hypervisor-based integrity measurement agent. In *2009 Annual Computer Security Applications Conference*, pages 461–470, Dec 2009.

[7] N. Z. Bawany, J. A. Shamsi, and K. Salah. DDoS Attack Detection and Mitigation Using SDN: Methods, Practices, and Solutions. *Arabian Journal for Science and Engineering*, 42:425–441, 2017.

[8] C. Bihary. Unveiling the true cost of software-defined networking. https://www.garlandtechnology.com/blog/unveiling-the-true-cost-of-software-defined-networking. Accessed: 2020-04-29.

[9] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. TOCTOU, Traps, and Trusted Computing. In P. Lipp, A.-R. Sadeghi, and K.-M. Koch, editors, *Trusted Computing - Challenges and Applications*, pages 14–32, 2008.

[10] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. Problems with the Static Root of Trust for Measurement. *Black Hat USA*, 2013.

[11] Carta.tech. Man pages in libauparse-dev - carta.tech. https://www.carta.tech/packages/libauparse-dev/. Accessed: 2020-05-05.

[12] cisco corporation. Nexus 9000 tcam carving. https://www.cisco.com/c/en/us/support/docs/switches/nexus-9000-series-switches/119032-nexus9k-tcam-00.html. Accessed: 2020-04-29.

[13] M. Corporation. Windows filtering platform. https://docs.microsoft.com/en-us/windows/win32/fwp/windows-filtering-platform-start-page. Accessed: 2020-04-01.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 254–265, 2011.

[15] Datanyze. Docker market share and competitor reporth. https://www.datanyze.com/market-share/containerization--321/docker-market-share. Accessed: 2020-05-05.

[16] Dell. Dell networking 8132f. http://www.netsolutionworks.com/datasheets/Dell_PowerConnect_8100_Series_Spec_Sheet.pdf. Accessed: 2020-04-29.

[17] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE symposium on security and privacy*, pages 297–312. IEEE, 2011.

[18] J. Edge. Audit, namespaces, and containers. https://lwn.net/Articles/699819/. Accessed: 2020-05-01.

[19] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp. Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks. In *Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference*, pages 1–8, 2012.

[20] O. N. Foundation. Openflow 1.0 switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf. Accessed: 2020-03-31.

[21] O. N. Foundation. Openflow 1.3 switch specification. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf. Accessed: 2020-03-31.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206, 2003.

[23] T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206. Citeseer, 2003.

[24] graphviz.org. The dot language. https://www.graphviz.org/doc/info/lang.html. Accessed: 2020-04-20.

[25] I. . W. Group. 802.1q - virtual lans. http://www.ieee802.org/1/pages/802.1Q.html. Accessed: 2020-04-01.

[26] R. Hat. Chapter 7. system auditing. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing. Accessed: 2020-05-01.

[27] Hewlett-Packard. Aruba 5400 zl switch series - specifications. https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c01814551. Accessed: 2020-04-29.

[28] Hewlett-Packard. Hp sdn hybrid network architecture. https://community.arubanetworks.com/aruba/attachments/aruba/SDN/43/1/4AA5-6738ENW.PDF. Accessed: 2020-04-29.

[29] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 43–48, New York, NY, USA, 2013. Association for Computing Machinery.

[30] X. Jiang and X. Wang. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In C. Kruegel, R. Lippmann, and A. Clark, editors, *Recent Advances in Intrusion Detection*, pages 198–218, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[31] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM.

[32] P. Kampanakis, H. Perros, and T. Beyene. Sdn-based solutions for moving target defense network protection. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6, 2014.

[33] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16, 2016.

[34] A. P. Kosoresow and S. Hofmeyer. Intrusion detection via system call traces. *IEEE software*, 14(5):35–42, 1997.

[35] M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about sdn flow tables. In *International Conference on Passive and Active Network Measurement*, pages 347–359. Springer, 2015.

[36] M. Kuzniar, P. Peresini, and D. Kostic. What you need to know about sdn flow tables. *Lecture Notes in Computer Science (LNCS)*, 2015.

[37] Y. Lei and C. A. Shue. Detecting root-level endpoint sensor compromises with correlated activity. In *SecureComm*, 2019.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, 2008.

[39] B. Min and V. Varadharajan. A Novel Malware for Subversion of Self-Protection in Anti-virus. *Softw. Pract. Exper.*, pages 361–379, 2016.

[40] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings 13th Annual Computer Security Applications Conference*, pages 175–181. IEEE, 1997.

[41] S. M. Mousavi and M. St-Hilaire. Early detection of ddos attacks against sdn controllers. In *2015 International Conference on Computing, Networking and Communications (ICNC)*, pages 77–81, 2015.

[42] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 18, USA, 2010. USENIX Association.

[43] M. E. Najd and C. A. Shue. Deepcontext: An openflow-compatible, host-based sdn for enterprise networks. In *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, pages 112–119, 2017.

[44] S. G. O. M. Paul Moore, Richard Guy Briggs. linux-audit/audit-userspace. `https://github.com/linux-audit/audit-userspace`. Accessed: 2020-05-05.

[45] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *1 USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[46] O. Project. Openwrt project: Welcome to the openwrt project. `https://openwrt.org/`. Accessed: 2020-04-01.

[47] D. Raumer, L. Schwaighofer, and G. Carle. Monsamp: A distributed sdn application for qos monitoring. In *2014 Federated Conference on Computer Science and Information Systems*, pages 961–968, 2014.

[48] S. W. Smith. Secure Coprocessor. In J. S. van Tilborg, Henk C. A., editor, *Encyclopedia of Cryptography and Security*, pages 1102–1103. Springer US, 2011.

[49] M. Soysal and E. G. Schmidt. Machine learning algorithms for accurate flow-based network traffic classification: Evaluation and comparison. *Performance Evaluation*, 67(6):451 – 467, 2010.

[50] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. Past: Scalable ethernet for data centers. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, page 49–60, New York, NY, USA, 2012. Association for Computing Machinery.

[51] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho. Deep learning approach for network intrusion detection in software defined networking. In *2016 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 258–263, 2016.

[52] C. R. Taylor. Leveraging software-defined networking and virtualization for a one-to-one client-server model. *Masters These*, 557, 2014.

[53] C. R. Taylor, D. C. MacFarland, D. R. Smestad, and C. A. Shue. Contextual, Flow-Based Access Control with Scalable Host-Based SDN Techniques. *IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[54] M. Thottan, L. Li, B. Yao, V. S. Mirrokni, and S. Paul. Distributed network monitoring for evolving ip networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 712–719, 2004.

[55] Trusted Computing Group. Trusted Platform Module 2.0: A Brief Introduction, 2019. [Online; accessed 10-April-2019].

[56] Ubuntu.com. Apparmor- ubuntu wiki, 2019. [Online; accessed 18-Nov.-2019].

[57] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[58] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen. Scotch: Elastically Scaling up SDN Control-Plane Using vSwitch Based Overlay. In *ACM International on Conference on Emerging Networking Experiments and Technologies*, pages 403–414, 2014.

[59] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu. Ruletris: Minimizing rule update latency for tcam-based sdn switches. In *International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[60] T. Y. Win, H. Tianfield, and Q. Mair. Detection of malware and kernel-level rootkits in cloud computing environments. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, pages 295–300, Nov 2015.

[61] P. Winter. An algorithm for the enumeration of spanning trees. *BIT*, 26(1):44–62, Jan. 1986.

[62] Y. Xu and Y. Liu. Ddos attack detection under sdn context. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[63] A. Zaalouk, R. Khondoker, R. Marx, and K. M. Bayarou. Orchsec: An orchestrator-based architecture for enhancing network-security using network monitoring and sdn control functions. In *NOMS*, pages 1–9, 2014.