Detecting Root-Level Endpoint Sensor Compromises with Correlated Activity

Yunsen Lei and Craig A. Shue*

Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA, USA
{ylei3, cshue}@wpi.edu

Abstract. Endpoint sensors play an important role in an organization's network defense. However, endpoint sensors may be disabled or sabotaged if an adversary gains root-level access to the endpoint running the sensor. While traditional sensors cannot reliably defend against such compromises, this work explores an approach to detect these compromises in applications where multiple sensors can be correlated. We focus on the OpenFlow protocol and show that endpoint sensor data can be corroborated using a remote endpoint's sensor data or that of in-network sensors, like an OpenFlow switch. The approach allows end-to-end round trips of less than 20ms for around 90% of flows, which includes all flow elevation and processing overheads. In addition, the approach can detect flows from compromised nodes if there is a single uncompromised sensor on the network path. This approach allows defenders to quickly identify and quarantine nodes with compromised endpoint sensors.

Keywords: Endpoint security · Compromise detection · Software-defined networking

1 Introduction

To protect their organizations, defenders typically deploy a mixture of perimeter defenses and defense-in-depth techniques, such as endpoint tools and sensors. Perimeter and endpoint defenses have varying strength: perimeter defenses can be centrally administered and have a global view while endpoint sensors often have detailed context about the associated endhost and can detect intra-subnet traffic. End-user activity on the endpoint represents a common vector for threats to enter an organization [13]. The value of context around these user actions has caused even leaders of perimeter-based defenses to begin creating endpoint sensors [15, 3].

While endpoint sensors can play an important role, they come with a significant risk: an adversary could compromise the sensor to silence its reporting or to provide false information. Many endpoint solutions, such as firewalls, host-based intrusion detection systems, and anti-virus, rely upon a least-privilege assumption

* Shue holds stock in ContexSure Networks, Inc., an arrangement that has been reviewed and approved by WPI's Conflict Management Committee.

in which the end-user operates in a regular user account. Those solutions further assume that processes running as an administrator user and all kernel space functionality will remain uncompromised. In the event of a root-level compromise, these tools may use anti-circumvention techniques to hinder their removal or sabotage, but they ultimately cannot offer any security guarantees.

In some cases, the data provided by endpoint sensors can be corroborated with data from other sensors in the network or remote endpoints. A sensor's reports about network flows, for example, can be easily corroborated by examining the data flows reported by sensors on other machines. In some instances, all of the report data may be corroborated while in other instances, only a portion of the data can be corroborated. For example, a remote network IDS could verify that the network layer packet headers are accurate, but 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 paper, we examine how an endpoint sensor providing data via the OpenFlow protocol can be evaluated using data from other OpenFlow nodes. In the OpenFlow protocol, an OpenFlow agent seeks guidance from a logically centralized controller whenever the agent encounters a packet for which it lacks a matching flow rule in its cache. When these caches are empty or when fine-grained (i.e., connection-specific) flow rules are used, each OpenFlow agent along the new flow's path will "elevate" a request to the controller. Accordingly, a controller can receive multiple reports about each new connection and determine if the OpenFlow agent requests are consistent. If it detects any inconsistency, the controller may be able to pinpoint a compromised or faulty node.

In this paper, we make the following contributions:

- **Implement an Endpoint Flow Verification System:** In our Correlated Host-based OpenFlow Sensor Enforcement (CHOSE) system, an endpoint sensor reports flow and contextual data (e.g., originating user and application) for each new network connection. We correlate these reports across endpoint and in-network devices to detect potentially compromised endpoints.
- **Evaluate the System's Security and Performance:** Our system can detect and block flows with inaccuracies that indicate a compromised sensor even before a flow is fully established. We further find that legitimate flows can corroborated by remote sensors and complete their first round trip in less than 20 milliseconds for about 90% of flows. We find that a controller can easily detect a faulty OpenFlow agent when a single non-compromised sensor is on the flow's path.

2 Background and Related Work

In this section, we provide a brief overview of the OpenFlow protocol and its use. We then describe prior work related to host-based software-defined networking (SDN), detecting attacks using SDNs, and the detection of endpoint compromises.

2.1 OpenFlow and Software-Defined Networking (SDN)

In the software-defined networking paradigm, control-plane decisions are separated from the underlying hardware that performs packet forwarding. The OpenFlow protocol [9] provides an API for a logically centralized controller to interact with a set of packet forwarding devices, which are often network switches. In OpenFlow, a switch will send a `PacketIn` packet to an OpenFlow controller whenever the switch 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 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 indicates what the controller should do with the packet contained in the `PacketIn` message.

The OpenFlow protocol allows a controller to essentially treat each OpenFlow switch as a configurable rule cache. A controller could proactively 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 will only match a single connection. 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.

2.2 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) [16] is often used on virtual machine (VM) hypervisors to provide SDN functionality between VMs. In the Scotch approach, Wang et al. [23] proposed using OVS to enhance the scalability of fine-grained flows by using OVS on VM hypervisors.

Taylor et al. [20] proposed a host-based SDN that provides information about the end host in addition to the network flow information. Najd and Shue [12] 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. These latter two host-based SDNs fall into the class of endpoint sensors that we focus on in this paper.

2.3 Detecting Attacks with SDN

SDNs can provide both centralized and fine-grained flow-based network access control. Prior work has examined how to leverage these features to improve the network's security.

Jafarian et al. [7] proposed a technique called OpenFlow Random Host Mutation (OF-RHM) which use OpenFlow to mutates hosts' IP addresses frequently and randomly, consequently prevent adversaries from accurately identifying the target. T. Xing et al. proposed SnortFlow [24] which integrates OpenFlow and Snort [17] to support efficient and flexible Intrusion Prevention Systems (IPS). To better detect attacks from inside the internal network, Shin and Gu proposed CloudWatcher [18] which leverages OpenFlow to control flows and direct them through a cloud-based middlebox for security inspection. Rodrigo et al. [4] combined OpenFlow and Self Organizing Map (SOM) [8], a neural network trained to perform traffic classification. They use SOM to classify network traffic as normal or abnormal and rely on Nox controller to control the flows. Finally, Bawany et al. [1] proposed an SDN-based proactive DDoS Defense Framework (ProDefense) to detect and mitigate the consequence of such attackers for data centers.

Our approach differs from these types of systems by leveraging end-point sensors and verifying the accuracy of the data reported by these sensors, even if the underlying host is compromised.

2.4 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. As an example, malware has been found to deactivate anti-virus [11] to evade detection. Attackers may also disguise their traffic, using mimicry techniques [22], to appear legitimate.

To relax assumptions about a trustworthy OS, trusted hardware, such as trusted platform modules (TPMs) [21] or secure co-processors [19], can be used to provide attestations. These approaches tend to suffer from fragility to minor changes (when a static root of trust is used) [6] or from classic time-of-check-time-of-use (TOCTOU) issues [5] (when a dynamic root of trust is used).

In this work, we proceed in a different direction: we try to detect sensors that provide inaccurate data, or omit data, by comparing their outputs with other sensors on the network. This distributed monitoring approach can highlight attacks even without special trusted hardware.

3 Correlated Host-Based OpenFlow Sensor Enforcement

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

3.1 Example Endpoint Sensor Compromises

An organization may use endpoint firewalls or host-based intrusion detection systems (HIDS) in order to provide defense-in-depth protections. The organization

may configure each endpoint with a set of firewall and HIDS rules that must be enforced to detect and prevent the spread of attacks or data ex-filtration. Unfortunately, one host may be compromised by an attack that gets through the firewall (e.g., a malicious email attachment) or in an out-of-band manner (e.g., through an infected USB device). In some cases, the compromise may occur at the administrator or root level due to the end-user running with administrator privileges or due to a privilege escalation attack.

Without additional network sensors, an adversary could simply disable the endpoint firewall or IDS to engage in arbitrary network communication and to prevent event reporting. Without additional enforcement or monitoring points, the attack would be successful.

With correlated sensing, other network sensors could detect the disallowed communication. A network gateway collecting sampled netflow data [2], for example, could provide snapshots to a centralized IDS. If those flow samples contained flows that should have been blocked by an endpoint firewall, the centralized IDS could determine the firewall was faulty. Likewise, a host on the network receiving an illicit flow from another machine on the network that should be running a firewall that would block the connection could report the issue.

In some cases, sensors may be redundant (e.g., at both endpoints) while in other cases, they may have correlated behavior (e.g., IDSes and netflow records). Both types of data can help a defender detect inconsistencies that belie a host compromise.

3.2 System Overview and Threat Model

In Figure 1, 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 OpenFlow controller manages the OpenFlow 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.

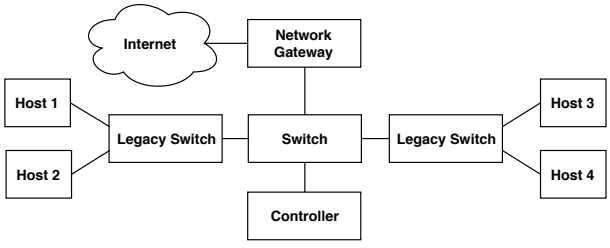

Fig. 1. An example enterprise network with OpenFlow agents on each end-point.

In this example, the trusted computing base (TCB) includes the OpenFlow controller and the network switches. The physical connections between the

switches, hosts, and controllers are considered uncompromised and reliable. The hosts on the network are not part of the TCB; the sensor data obtained from these hosts could be erroneous or absent due to a root-level compromise on a host. In some cases, a set of compromised hosts may collude with a goal of evading detection. In other cases, a compromised host may communicate with an uncompromised host. In that case, the uncompromised host will report the communication. If any switch is an OpenFlow switch, it is in the TCB and it is managed by the controller, so it can be configured so that any communication through the OpenFlow switch will be reported to the controller.

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 [1]. 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.

3.3 Corroborated Sensing Deployment Scenarios

Some organizations deploy specialized security middleboxes, such as firewalls or IDSes, that can vet communication. Often, these middleboxes are deployed at network perimeters and they do not inspect internal traffic, such as intra-subnet flows. These organizations may deploy endpoint sensors to gain insight into intra-subnet traffic. But, with root-level compromises on the endpoints, these sensors may fail to produce complete or accurate data. With sensors at both endpoints, a network operator is more likely to detect a compromised sensor.

Using the network in Figure 2 as an example, consider a TCP `SYN` packet sent by Host 1 to Host 2. If both Host 1 and Host 2 provide OpenFlow sensor data, the controller will receive independent reports of this `SYN` packet within `PacketIn` elevations from these hosts (shown by lines 1 and 2 for Host 1 and lines 5 and 6 for Host 2). In this case, if either Host 1 or Host 2 provided inaccurate information about the `SYN` packet, or neglected to engage in a `PacketIn` elevation entirely, the controller will easily be able to detect the 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 OpenFlow endpoint sensor 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 sensor.

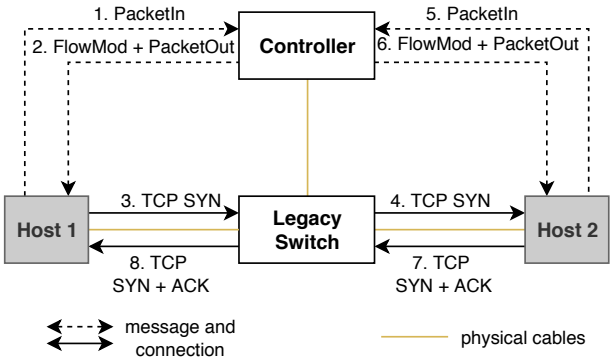

Fig. 2. When both endpoints run an OpenFlow agent, if either is uncompromised, that uncompromised sensor will alert the central coordinator of inconsistencies via its `PacketIn` data.

In the Figure 2 example, the controller will receive conflicting information and know one of the two hosts is compromised, but will not know which has the error. However, if the switch is an OpenFlow switch, as shown in Figure 3, the controller can determine which host is deceptive. The OpenFlow switch would provide information about the `SYN` packet (shown by lines 4 and 5). Further, since the OpenFlow 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.

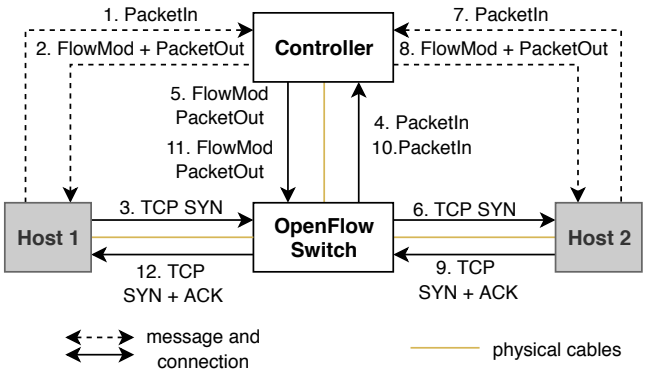

Fig. 3. When an OpenFlow switch is on the network path, the controller receives `PacketIn` data that allows it to identify which endpoint, if any, is faulty.

When using corroborated sensing, particularly when only the endpoints have sensors (e.g., Figure 2), the controller must be careful in how it manages the

rules it stores at each endpoint. By pushing uni-directional flow rules in its initial `FlowMod` messages, the controller can detect if the destination fails to properly elevate traffic. In the `FlowMod` messages in Message 2 in Figure 2 and Messages 2 and 5 in Figure 3, the controller only pushes an approval for the flow in the direction from Host 1 to Host 2. When responding to the destination, and on agents elevating the `SYN+ACK` packets, the controller orders the agents to store a bi-directional `FlowMod` approving both directions of the flow.

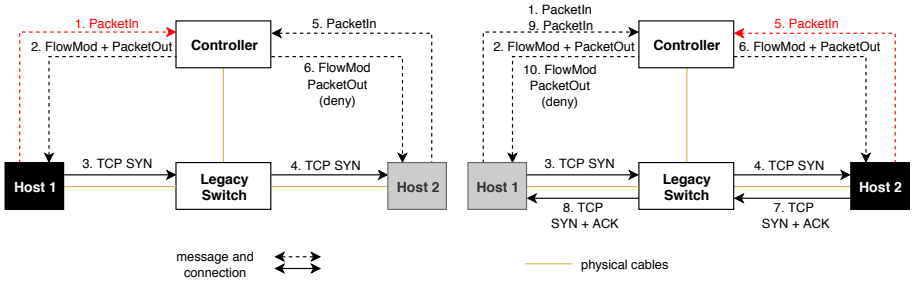

Fig. 4. When one of the hosts is compromised (shaded in black), the controller will notice a discrepancy when receiving a `PacketIn` from the non-compromised host (shaded in gray).

In Figure 4, 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 4), 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 4, 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.

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 OpenFlow switch that connects the devices (as shown in Figure 5), this scenario cannot be avoided.

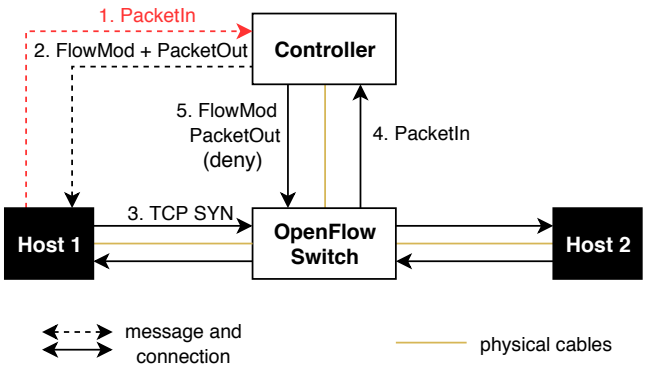

Fig. 5. If both hosts collude (shaded in black), only a middlebox or an OpenFlow switch between the hosts can be used to detect the malicious flows.

3.4 Uncorroborated Data in Endpoint Sensors

In the OpenFlow data, all sensor data can be corroborated since the only information, the elevated packet, is included it its entirety and is independently witnessed by multiple vantage points. However, other sensors may include data that is only available at a single vantage point.

The host-based SDNs created by Taylor et al. [20] and Najd et al. [12] provide additional information about the network flows. Some of that information includes the user account and originating application on a sending endpoint and the destination server and its user on a receiving endpoint. Since this context is only available on the respective endpoints, neither the other endpoint nor a middlebox can corroborate that contextual data. Accordingly, a compromised host could arbitrarily forge this contextual data.

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. However, a sophisticated adversary would likely be able to craft contextual data that would plausibly be associated with the verifiable network headers and packet payload.

Some endpoint sensors, such an a reporting engine for an anti-virus tool, may engage in communication that is completely unverifiable by other sensors. These sensors would not be able to effectively use correlated sensing on its own.

To gain trust in information that cannot be corroborated, trusted hardware or VM introspection techniques may play a role. However, in situations where corroboration is possible, correlated sensing can provide benefits without requiring special hardware.

4 Implementing the CHOSE System

The CHOSE system has three components: 1) a standard OpenFlow agent for physical switches, 2) an OpenFlow-compatible host agent for Microsoft Windows machines, and 3) a custom OpenFlow controller that manages connections for both switches and end-hosts. For the OpenFlow agent, we use the built-in OpenFlow agent on an enterprise-grade switch. In the remainder of this section, we focus on the host agent and the functionality in the OpenFlow controller.

4.1 Host Agent for Microsoft Windows

While OpenFlow implementations are available in VM hypervisors, such as through OVS [16] or HyperV[10], these implementations are not designed for end-user systems. Recent host-based SDNs have focused on obtaining contextual information about the host's operation, rather than simply providing OpenFlow functionality on an endpoint [20, 12]. These tools are particularly valuable on end-user machines, like desktops and laptops, where the end-user's actions take place. However, these prior SDN systems have been implemented in the Linux operating system, which constitutes roughly 2-3% of the desktop and laptop market share. In contrast, Microsoft Windows has roughly 86-88% of the desktop and laptop market share [14]. To have a major impact on how systems are used, we need to focus on an endpoint solution for Microsoft Windows.

We created a host-based SDN agent for Microsoft Windows using a kernel-mode Windows driver. The driver uses the Windows Application Layer Enforcement (ALE) portion of the Windows Filtering Platform (WFP) to monitor all socket operations, including the creation of TCP and UDP connections. The ALE filtering approach allows us to monitor traffic at a per-connection or per-socket level rather than having to process packets individually, allowing us to recreate the OpenFlow process natively in Windows.

The SDN agent communicates to an SDN controller using a modification to the OpenFlow protocol. As with standard OpenFlow, the agent elevates a packet by including an OpenFlow header and encapsulating a copy of the original packet. However, the SDN agent also includes contextual information about the application in a custom structure that follows the encapsulated packet. This contextual information includes the application path of the sending application, the user running the software, and the process identifier among other fields. The OpenFlow communication is then encrypted and authenticated using AES encryption and a SHA-256 message authentication code (MAC).

Upon receiving a response from the SDN controller, the host-based agent either drops the packet (for discard decisions) or updates the flow status in the WFP framework and reinjects the packet into the network kernel queue for delivery.

The SDN agent requires cooperation from the Windows kernel and from an administrative service on the machine. The kernel intercepts the network communication and delivers it to the administrative service. The service is responsible for the network communication and for the gathering of the process

context. Typically, user space applications and the kernel interact through system calls. However, in this case, the kernel initiates the flow elevation requests based on the detection of new flows, so a different communication model is needed. We use an inverted call model using a device driver queue shared between the kernel and user space to facilitate this interaction.

4.2 OpenFlow Controller Customization

The SDN controller must support both the Windows OpenFlow agent and communication from traditional OpenFlow agents running on switches. This controller distinguishes the OpenFlow 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 OpenFlow agents would be on the path from the source machine to the destination for that flow. If the `PacketIn` arrives from the first expected OpenFlow 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 OpenFlow agent to approve the source to destination direction of the flow.

Since the controller sent a `FlowMod` only to the originating OpenFlow agent during its approval, subsequent OpenFlow agents on the path will again elevate the packet to the controller. If the controller receives a `PacketIn` from an OpenFlow 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 OpenFlow 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 OpenFlow 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.

5 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 correlated sensing introduce to an existing SDN deployment? 2) What security guarantees does correlated sensing offer to such a system?

5.1 Experiment Setup

In both our performance and security evaluation, we configure our network to match Figure 3. We use an HP 2920-24G enterprise switch with OpenFlow enabled to connect our hosts and controller. Our controller runs on a laptop that runs VirtualBox to host an Ubuntu 16.04 VM. We configure the controller with 2 IP address and place one of the IP addresses under the control of OpenFlow so that the communication between sensors and controller will also be subject to the OpenFlow'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.

5.2 Performance Evaluation

To determine the overhead associated with correlated sensing, we compare it with regular OpenFlow 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 OpenFlow 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 OpenFlow Only:** In this scenario, neither of the hosts run an OpenFlow agent and simply transmit packets using the native Windows networking stack. The physical OpenFlow switch elevates each new connection it sees to the OpenFlow 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 Sensors Only:** In this scenario, both of the hosts run our Windows OpenFlow 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 OpenFlow switch is configured as a simple learning switch and does not send any elevation requests to the controller.
- **Scenario 3: Both Switch and Host Sensors:** 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 OpenFlow agents.
- **Scenario 4: Full Sensing and Flow Correlation:** In this scenario, the OpenFlow agents run at the hosts and the physical switch. The controller examines the elevations across OpenFlow agents and correlates the requests to identify discrepancies or missing elevation requests.

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 6.

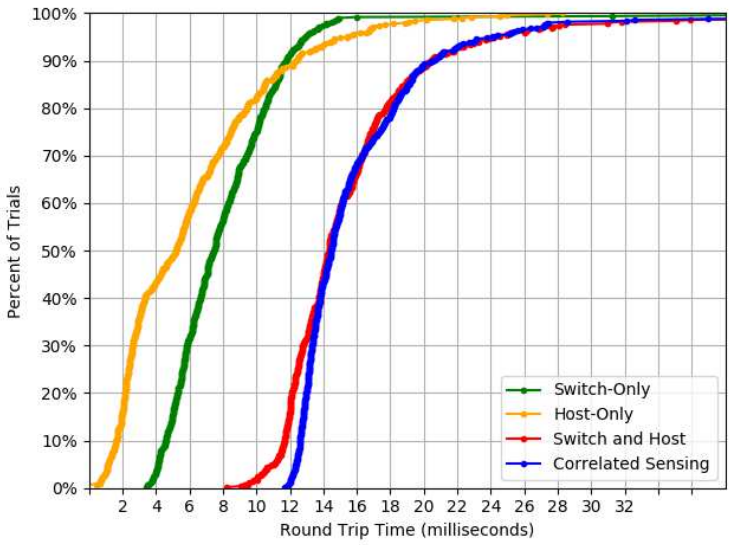

Fig. 6. Round-trip time of serial connections across 500 trials.

The overhead of the correlated sensing is the timing difference between the third scenario, in which host and switch sensors are used but the controller acts statelessly, and the fourth scenario, in which the sensors are identical but the controller correlates flows across OpenFlow agents. In Figure 6, the distribution curve of the round-trip times associated with these two scenarios largely overlap, indicating that the performance costs of correlated sensing are not significant.

From these experiments, we see that most flows complete in less than 15 milliseconds, even with correlated 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 in software whereas it can use a hardware table for subsequent packet forwarding once a `FlowMod` is installed. 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.

Parallel Connections We next examine the performance of the SDN approaches in a more real-world setting, we use end-to-end timing of a short HTTP connection. Using parallel threads, we generate HTTP requests on Host 1 to an HTTP server running on Host 2. The client application on Host 1 is designed to use a separate TCP connection for each HTTP request. It issues an HTTP `GET` request for a short HTML document. After the server provides the HTML document, the client closes the connection. Using a varying thread count on the client, we measure how many new connections can be created by the client in a five minute (300 second) period.

Table 1. Number of flows created in five minute period.

Number of Threads	Scenario 1: Switch-Only	Scenario 2: Host-Only	Scenario 3: Switch and Host	Scenario 4: Full Correlation	Scenario 5: Host Correlation
10	2,919	2,965	2,813	2,365	2,830
20	5,431	5,520	4,525	4,519	5,221
50	13,440	13,782	10,052	9,214	13,054
100	23,841	23,298	16,389	16,194	19,926

In Table 1, we show the results of the parallel connection experiments. We see that in the host-only scenario, the hosts and controller can handle an average of roughly 74 new flows per second. We create a fifth scenario that uses correlation only at the endpoints sensors, called Scenario 5, and see that it largely keeps pace with Scenario 2 up to 50 concurrent threads, but starts to slow down at 100 concurrent threads. That likely indicates that controller bottlenecks begin to form at that higher thread count.

Additionally, when looking at Scenarios 3 and 4 in Table 1, we note a marked decrease in the number of new flows during the testing period compared to the other cases. In essence, it appears that these scenarios were latency bound: each

thread had to spend more time in elevations because there were serial elevations for each thread from the hosts and the switch. In Table 2, we see that the number of OpenFlow messages required (namely `PacketIn` messages and the associated `PacketOut+FlowMod` responses) are increased by a factor of three in Scenarios 3 and 4 because the hosts and the switch are each performing the elevations and the switch is also elevating the end-hosts' own `PacketIn` messages. As a result, each thread simply spends more time waiting for the initial round-trip.

Table 2. Number of OpenFlow messages in each scenario.

	Elevation Source			Total
Scenario	Host 1	Switch	Host 2	Messages
1	0	4	0	4
2	2	0	2	4
3	2	8	2	12
4	2	8	2	12
5	2	0	2	4

5.3 Security Evaluation

We examine the effectiveness of the correlated 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 OpenFlow 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 3.3, if the switch between the hosts is legacy, the uncompromised host triggers the controller's detection rather than the OpenFlow 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.

6 Conclusion

In this work, we examine how network operators can detect even root-level compromises that affect the accuracy of data reported by host-based sensors by correlating that data with other sensors in the network. We focused on the OpenFlow protocol and showed 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 correlated 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.

7 Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1422180.

References

[1] Bawany, N.Z., Shamsi, J.A., Salah, K.: DDoS Attack Detection and Mitigation Using SDN: Methods, Practices, and Solutions. Arabian Journal for Science and Engineering **42**, 425–441 (2017). https://doi.org/10.1007/s13369-017-2414-5

[2] Berthier, R., Cukier, M., Hiltunen, M., Kormann, D., Vesonder, G., Sheleheda, D.: Nfsight: Netflow-Based Network Awareness Tool. In: Large Installation System Administration Conference. p. 119 (2010)

[3] Bhattarai, R., Valle, E., Dhanraj, M., Kelly, R.: Advanced Endpoint Protection Test Report. Tech. rep., Palo Alto Networks (2018), `https://www.paloaltonetworks.com/resources/whitepapers/2018-nss-labs-advanced-endpoint-protection-report`

[4] Braga, R., Mota, E., Passito, A.: Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow. In: IEEE Local Computer Network Conference. pp. 408–415 (2010). https://doi.org/10.1109/LCN.2010.5735752

[5] Bratus, S., D'Cunha, N., Sparks, E., Smith, S.W.: TOCTOU, Traps, and Trusted Computing. In: Lipp, P., Sadeghi, A.R., Koch, K.M. (eds.) Trusted Computing - Challenges and Applications. pp. 14–32 (2008). https://doi.org/10.1007/978-3-540-68979-9_2

[6] Butterworth, J., Kallenberg, C., Kovah, X., Herzog, A.: Problems with the Static Root of Trust for Measurement. Black Hat USA (2013)

[7] Jafarian, J.H., Al-Shaer, E., Duan, Q.: Openflow Random Host Mutation: Transparent Moving Target Defense Using Software Defined Networking. In: Workshop on Hot Topics in Software Defined Networks. pp. 127–132 (2012). https://doi.org/10.1145/2342441.2342467

[8] Kohonen, T.: The Self-Organizing Map. Proceedings of the IEEE **78**, 1464–1480 (1990). https://doi.org/10.1109/5.58325

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

[10] Microsoft: Hyper-V Architecture (2018), `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture`, [Online; accessed 09-April-2019]

[11] Min, B., Varadharajan, V.: A Novel Malware for Subversion of Self-Protection in Anti-virus. Softw. Pract. Exper. pp. 361–379 (2016). https://doi.org/10.1002/spe.2317

[12] Najd, M.E., Shue, C.A.: Deepcontext: An Openflow-Compatible, Host-Based SDN for Enterprise Networks. In: IEEE Conference on Local Computer Networks (LCN). pp. 112–119 (2017). https://doi.org/10.1109/LCN.2017.12

[13] Neely, L.: Exploits at the Endpoint: SANS 2016 Threat Landscape Survey. SANS Institute InfoSec Reading Room, September (2016), `https://www.sans.org/reading-room/whitepapers/firewalls/paper/37157`

[14] Net Marketshare: Market Share Statistics for Internet Technologies (2019), https://netmarketshare.com/operating-system-market-share.aspx, [Online; accessed 10-April-2019]
[15] Palo Alto Networks: Traps Technology Overview. Tech. rep., Palo Alto Networks (February 2019), https://www.paloaltonetworks.com/resources/techbriefs/traps-technology-overview
[16] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., Casado, M.: The Design and Implementation of Open vSwitch. In: 1 USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 117–130 (2015), https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff
[17] Roesch, M., et al.: Snort: Lightweight Intrusion Detection for Networks. In: LISA. vol. 99, pp. 229–238 (1999)
[18] Seungwon Shin, G.G.: Cloudwatcher: Network Security Monitoring Using Openflow in Dynamic Cloud Networks (or: How to Provide Security Monitoring as a Service in Clouds?). In: IEEE International Conference on Network Protocols (ICNP). pp. 1–6 (2012). https://doi.org/10.1109/ICNP.2012.6459946
[19] Smith, S.W.: Secure Coprocessor. In: van Tilborg, Henk C. A., J.S. (ed.) Encyclopedia of Cryptography and Security, pp. 1102–1103. Springer US (2011)
[20] Taylor, C.R., MacFarland, D.C., Smestad, D.R., Shue, C.A.: Contextual, Flow-Based Access Control with Scalable Host-Based SDN Techniques. IEEE International Conference on Computer Communications pp. 1–9 (2016)
[21] Trusted Computing Group: Trusted Platform Module 2.0: A Brief Introduction (2019), https://trustedcomputinggroup.org/resource/trusted-platform-module-2-0-a-brief-introduction/, [Online; accessed 10-April-2019]
[22] Wagner, D., Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems. In: ACM Conference on Computer and Communications Security. pp. 255–264 (2002). https://doi.org/10.1145/586110.586145
[23] Wang, A., Guo, Y., Hao, F., Lakshman, T., Chen, S.: Scotch: Elastically Scaling up SDN Control-Plane Using vSwitch Based Overlay. In: ACM International on Conference on Emerging Networking Experiments and Technologies. pp. 403–414 (2014). https://doi.org/10.1145/2674005.2675002
[24] Xing, T., Huang, D., Xu, L., Chung, C., Khatkar, P.: Snortflow: A openflow-based intrusion prevention system in cloud environment. In: 2013 Second GENI Research and Educational Experiment Workshop. pp. 89–92 (2013)