SPHINX: Detecting Security Attacks in Software-Defined Networks

Mohan Dhawan
IBM Research
mohan.dhawan@in.ibm.com

Rishabh Poddar
IBM Research
rishabh.poddar@in.ibm.com

Kshiteej Mahajan
IBM Research
kshiteej.mahajan@in.ibm.com

Vijay Mann
IBM Research
vijaymann@in.ibm.com

***Abstract*—Software-defined networks (SDNs) allow greater control over network entities by centralizing the control plane, but place great burden on the administrator to manually ensure security and correct functioning of the entire network. We list several attacks on SDN controllers that violate network topology and data plane forwarding, and can be mounted by compromised network entities, such as end hosts and soft switches. We further demonstrate their feasibility on four popular SDN controllers. We propose SPHINX to detect both known and potentially unknown attacks on network topology and data plane forwarding originating within an SDN. SPHINX leverages the novel abstraction of *flow graphs*, which closely approximate the actual network operations, to enable incremental validation of all network updates and constraints. SPHINX dynamically learns new network behavior and raises alerts when it detects suspicious changes to existing network control plane behavior. Our evaluation shows that SPHINX is capable of detecting attacks in SDNs in realtime with low performance overheads, and requires no changes to the controllers for deployment.**

I. INTRODUCTION

The value of Software-Defined Networks (SDNs) lies specifically in their ability to provide network virtualization, dynamic network policy enforcement, and greater control over network entities across the entire network fabric at reduced operational cost. Protocols like OpenFlow [35] focus specially on the above aspects. However, by centralizing the control plane, SDNs place great burden on the administrator to manually ensure security and correct functioning of the entire network. Compromised network entities can be used to exfiltrate sensitive information, implement targeted attacks on other users, or simply bring down the entire network. This paper looks at the specific problem of detecting security attacks on network topology and data plane forwarding originating within SDNs in realtime.

Most prior work has looked at development and analysis of SDN security applications and controllers [22], [25], [26], [36], [38], [41], [43], and realtime verification of network constraints [20], [21], [24], [28]–[30], [34] separately. However, no combination of the above solutions provide an effective defense against the threat of attacks in SDNs due to compromised end hosts or switches, which can be used to wrest control of the entire network or parts of it [31], [32]. This problem is further exacerbated in the SDN context due to four main reasons.

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23064

First, operational semantics of OpenFlow-based SDNs lower the barrier for mounting sophisticated attacks on both control and data planes, since they allow any unmatched packets to be sent to the controller (similar to how a layer-2 switch broadcasts all unknown packets). For example, the SDN controller propagates and builds network topology using the OpenFlow `PACKET_IN` messages. However, even end hosts can send forged messages that would be relayed to the controller as `PACKET_IN` messages by the switches, thereby poisoning its view of the network. Although OpenFlow supports optional TLS authentication between switch and controller, TLS by itself cannot prevent compromised switches from spoofing packets. Thus, there is no built-in security for SDNs (even with TLS enabled) that prevents malicious switches and hosts from packet spoofing to corrupt controller state.

Second, attacks that affect traditional networks may also afflict SDNs. However, solutions that work for traditional networks may not be directly applicable for SDNs because traditional defenses assume switches to be intelligent, whereas separation of control and data planes forces SDN switches to be dumb forwarding entities that forward packets based on the rules installed by the SDN controller. Adapting traditional defenses for SDNs will require either patching the controller for *specific* vulnerabilities, or a *fundamental* redesign of the OpenFlow protocol to provide a comprehensive defense, without which many traditional attacks, including ARP poisoning and LLDP spoofing, will continue to manifest in SDNs.

Third, enterprise network administrators often use programmable soft switches, like Open vSwitches [13] (or OVSes), to provide network virtualization. These OVSes, just like hardware switches, must have direct connectivity to the controller to provide desired functionality. Further, since these soft switches run atop end host servers, they are attractive targets for attackers. In contrast, in traditional networks, it is relatively more difficult for a network attacker to physically compromise hardware switches and modify routing rules that govern network communication. Thus, the assumption that all switches in an SDN are trustworthy does not hold true in enterprise deployments.

Fourth, apart from potentially malicious switches, even untrusted end hosts can easily bring down the entire network. End hosts can initiate control plane flooding which can saturate the out-of-band network and interrupt the controller, thereby bringing down the entire network.

We tested four popular controllers: Floodlight [17], Maestro [8], OpenDaylight (ODL) [14] and POX [16], and found them vulnerable to diverse attacks originating within the

SDN [1]. While it is possible for controllers to implement defenses against known attacks or specific vulnerabilities, such patching does not provide protection against unforeseen security threats. In this context, we present the design and implementation of SPHINX—a framework to detect attacks on network topology and data plane forwarding. SPHINX leverages the novel abstraction of *flow graphs*, which closely approximate the actual network operations, to (a) enable incremental validation of all network updates and constraints, thereby verifying network properties in realtime, and (b) detect both known and potentially unknown security threats to network topology and data plane forwarding without compromising on performance. SPHINX can also be deployed with minimal modifications to secure different controllers.

SPHINX analyzes specific OpenFlow control messages to learn new network behavior and metadata for both topological and forwarding state, and builds flow graphs for each traffic flow observed in the network. It continuously updates and monitors these flow graphs for permissible changes, and raises alerts if it identifies deviant behavior. SPHINX leverages custom algorithms that incrementally process network updates to determine in realtime if the updates causing deviant behavior should be allowed or not. SPHINX also provides a light-weight policy engine that enables administrators to specify expressive policies over network resources and detect security violations. Unlike today's controllers where each module implements its own checks making policy enforcement buggy, SPHINX provides a central point for enforcing complex policies.

We have built a controller agnostic prototype of SPHINX, which may even be implemented by SDN controllers as an application. We have evaluated SPHINX with both OpenDaylight and Floodlight controllers over a physical three-tiered network testbed and the Mininet network emulator [10]. SPHINX successfully detected *all* the attacks, with a sub-millisecond average detection time in presence of $1K$ hosts, and reported no false alarms with three diverse but benign real-world network traces [3], [4], [7]. We further evaluated SPHINX's performance with up to $10K$ Mininet hosts, which is representative of a small enterprise. SPHINX is capable of verifying $1K$ policies at every network update in just ~245μs, and imposes low CPU (~6%) and memory overheads (~14.5%) in the worst case.

This paper makes the following contributions:

(a) We examine four popular SDN controllers and demonstrate that they are vulnerable to a diverse array of attacks on network topology and data plane forwarding (§ III and § VIII).

(b) We present incremental flow graphs (§ IV) as a novel abstraction for realtime detection of security threats.

(c) We present the design and implementation of SPHINX (§ V and § VII) and its policy engine (§ VI), which allows network administrators to specify fine-grained security policies, and enables easy action attribution.

(d) We evaluate SPHINX to show that it is practical and involves acceptable overheads (§ IX-A and § IX-B). We also report on experiences gained using SPHINX in four different case studies (§ IX-C).

[1]Unless specified, SDNs imply OpenFlow-based SDNs.

II. BACKGROUND

SOFTWARE-DEFINED NETWORK (SDN). SDNs decouple network control and forwarding functions enabling (i) the network to become directly programmable, and (ii) the underlying infrastructure to be abstracted for applications and network services. Network intelligence is logically centralized in trusted software-based controllers that maintain a global view of the network of hosts, and commodity hardware and software switches, which are dumb forwarding entities.

OPENFLOW. The OpenFlow protocol defines commands and messages that enable the controller to interact with the forwarding plane. Every OpenFlow switch maintains a number of flow tables, with each table containing a set of flow entries. Each flow entry consists of (i) match fields against which incoming packets are compared, (ii) a set of instructions that define the actions to be performed on matched packets, and (iii) counters for flow statistics [15]. Further, a match field may either contain a specific value, or it may be wildcarded, indicating that all packets match against it regardless of value. When a switch receives a packet for which it has no matching entry, it sends the packet to the controller as a `PACKET_IN` message. The controller then decides how to handle the packet, and creates one or more flow entries in the switch using `FLOW_MOD` commands, directing the switch on how to handle similar packets in the future.

Other switch-to-controller messages that are relevant to this paper include `FEATURES_REPLY` and `STATS_REPLY` messages. The `FEATURES_REPLY` message notifies the controller of a switch's capabilities and port definitions. The controller builds its initial view of the network topology using these messages, and updates the view using certain `PACKET_IN` messages. `STATS_REPLY` messages communicate network statistics gathered at the switch per port, flow, and table (such as the total number of packets/bytes sent or received).

III. MOTIVATION

The correct functioning of an SDN requires that two key network properties—network topology and data plane forwarding—must always be preserved. In this section, we motivate the need for SPHINX, which can detect both known and potentially unknown security attacks on these two key SDN properties in realtime.

First, we describe two scenarios that are representative of the possible attacks on both the network topology and data plane forwarding, launched from compromised hosts and/or switches. While there can be other variants of these attacks, the mechanisms to poison the controller's view of the network primarily remain the same. Note that none of these attacks exploit any OpenFlow vulnerabilities or implementation bugs in particular controllers.

Second, we argue that traditional solutions to defend against known security threats in their exact form are not portable to SDNs. Any adaptations of these solutions to SDNs requires patching the controller. While it is possible for all controllers to implement defenses against known attacks or specific vulnerabilities, such selective signature-based security mechanisms suffer from the same issues that afflict anti-virus solutions and fail to protect against a broad class of malicious attacks possible on SDNs.

A. *Host- and Switch-based Attacks*

OpenFlow mandates that packets not matching a flow rule must be sent by the switch to the controller. In spite of the control and data plane separation, this protocol requirement opens up possibilities for malicious hosts to tamper with network topology and data plane forwarding, both of which are critical to the correct functioning of the SDN. Specifically, malicious hosts can (i) forge packet data that would then be relayed by the switches as `PACKET_IN` messages, and subsequently processed by the controller, (ii) implement denial of service (DoS) attacks on the controller and switches, and (iii) leverage side-channel mechanisms to extract information about flow rules. Compromised soft switches can not only initiate all the host-based attacks but also trigger dynamic attacks on traffic flows passing through the switch, resulting in (i) network DoS, and (ii) traffic hijacking or re-routing.

1) Network topology: SDN controllers process a variety of protocol packets (ARP, IGMP, LLDP, etc.) sent by switches as OpenFlow `PACKET_IN` messages to construct its view of the network topology. Controllers process LLDP messages for topology discovery and IGMP messages to maintain multicast groups, whereas it forwards ARP requests and replies enabling end hosts to build up ARP caches facilitating network communication. Compromised hosts can spoof the above messages to tamper with the controller's view of the topology, and fool it into installing flow rules to carry out a variety of attacks on the network.

EXAMPLE. A fake topology attack can be launched on an SDN controller to poison its view of the network using detrimental `PACKET_IN` messages sent by the switches. These malicious `PACKET_IN` messages could be generated by untrusted switches themselves or by end hosts, which can send arbitrary LLDP messages spoofing connectivity across arbitrary network links between the switches. When the controller tries to route traffic over these phantom links, it results in packet loss, and if this link is on a critical path, it could even lead to a blackhole.

2) Data plane forwarding: Malicious hosts and switches can mount DoS by flooding the network with traffic to arbitrary hosts to exhaust resources on vulnerable switches and/or the SDN controller, thereby affecting forwarding in the data plane.

EXAMPLE. TCAM is a fast associative memory that stores flow rules. Malicious hosts may target a switch's TCAM to perform directed DoS attacks against other hosts. Malicious hosts may send arbitrary traffic and force the controller into installing a large number of flow rules, thereby exhausting the switch's TCAM. Subsequently, no other flow rules can be installed on this switch, until the installed flows expire. If this switch is on a critical path in the network, then it may result in significant latency or packet drops.

In § VIII, we describe in detail several attacks, including those listed above, that afflict popular SDN controllers, like ODL, Floodlight, POX and Maestro.

B. *Traditional attacks manifest in SDNs*

Several attacks that afflict traditional networks also affect SDNs, where these attacks are triggered in part due to the intricacies of the SDN architecture, or the protocol involved (i.e., ARP, LLDP, etc.). However, adapting traditional defenses for these attacks in SDNs is non-trivial. This is because traditional networks often rely on switch intelligence to implement robust defenses against known security attacks. In contrast, SDN switches are mere forwarding entities without any intelligence. While patching SDN controllers to defend against known specific vulnerabilities is possible, it is not a comprehensive solution to detect all security attacks in SDNs.

EXAMPLES. In traditional networks, trustworthy verification of packets from neighboring switches to defend against LLDP spoofing requires cryptographic mechanisms, which is a heavyweight solution. In fact, message authentication amongst hosts and switches (even with TLS enabled) will not provide defense against corrupt routing rules in SDN switches, as is the case in the fake topology attack.

As another example, traditional networks defend against ARP poisoning either leveraging Dynamic ARP Inspection (DAI) [5], or requiring hosts to run programs like arpwatch [33] to set up static mappings. DAI mandates that switches snoop on all DHCP messages that pass through, and use that information to (i) prevent a rogue DHCP server from serving clients, and (ii) build a table of valid MAC to IP associations to validate ARP packets as they pass through. In contrast, SDN switches are dumb and cannot trivially extend DAI, while host based defenses are not comprehensive enough.

Both the above examples are representative of the fact that even simple and well-known defenses for attacks in traditional networks cannot be trivially extended to SDNs in a controller agnostic manner.

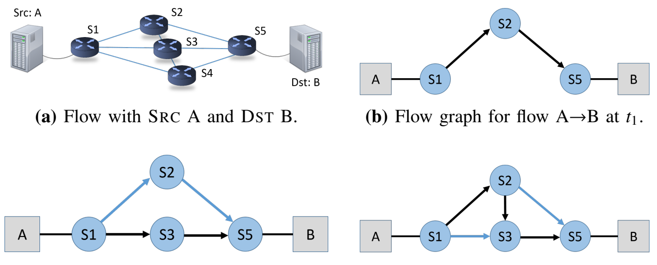
IV. SPHINX: OVERVIEW

A. *Threat model*

Attackers often break into the network to leverage internal vantage points, and subsequently launch attacks on the internal network. Since our goal is to (i) verify onset of attacks on network topology and data plane forwarding, and (ii) detect violations of policies *within* SDNs, our threat model focuses exclusively on scenarios where the adversary initiates attacks from within the SDN. Thus, we model SDNs as a closed system. Removing constraints on the *unknown* external communication helps focus our analysis only on OpenFlow control messages internal to the SDN.

We consider an enterprise SDN setup with no traffic across OpenFlow and non-OpenFlow network entities. We assume a trusted controller (which is required for the correct functioning of the network), but do not trust either the switches or the end hosts. This implies that the switches can lie about everything except their own identity, since the switches connect with the controller over separate TCP connections (possibly with TLS enabled). However, we do assume an honest majority of switches in the network. All prior art, including [28]–[30], [34], had assumed trustworthy switches, while SPHINX's threat model relaxes this requirement. Finally, given that most SDN applications run as modules as part of the controller binary, they can be trusted as long as the controller itself is trusted.

The assumptions above imply that OpenFlow communication from controller to switches is trustworthy, while from switches to controller is untrusted, and could be forged by a malicious switch or in some cases by hosts.



(a) Flow with Src A and Dst B. **(b)** Flow graph for flow A→B at t_1. **(c)** Flow graph for flow A→B at t_2. **(d)** Flow graph for flow A→B at t_3.

Fig. 1: Example flow, and construction of corresponding flow graph.

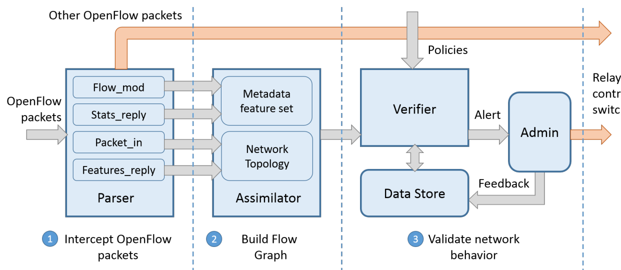


Fig. 2: Sphinx flow diagram.

B. Flow graphs

A *flow* is a directed traffic pattern observed between two endpoints with distinct MAC addresses over specified ports. A *flow graph* is a graph theoretic representation of a traffic flow with edges as the flow metadata and switches being the nodes in the graph. Sphinx uses these flow graphs to model both network topology and data plane forwarding in SDNs. It gleans flow metadata from OpenFlow control messages and *incrementally* builds the flow graphs to closely approximate the actual network operations, thereby enabling validation of all network updates and constraints on every flow graph in the network in realtime. Thus, flow graphs provide a clean mechanism that aids detection of diverse constraint violations for both network topology and data plane forwarding in SDNs.

Flow paths are constructed only using FLOW_MOD messages because they are issued by the trusted controller. Untrusted STATS_REPLY messages from each switch only update flow statistics of the corresponding switch, and do not affect the flow graph structure. Hence, the flow-specific network topology and data plane forwarding state as embodied in the flow graph remains uncorrupted even in the presence of untrusted switches and hosts. Further, as will be described later in § VI-B2, the presence of an honest majority of switches along the flow path enables Sphinx to precisely detect any malicious updates to flow statistics at any switch in the flow path.

As an example of the incremental construction of a flow graph, consider a flow between hosts A and B as shown in Figure 1a, that gets rerouted by the controller at different time steps. Figures 1b, 1c and 1d depict the state of the corresponding flow graph at each reroute, with the current path in black. The flow is first established at time-step t_1, with the path as $S1 \rightarrow S2 \rightarrow S5$. At t_2, the flow is rerouted by the controller along $S1 \rightarrow S3 \rightarrow S5$, and the current path is updated accordingly. Finally at t_3, the flow is rerouted once more along $S1 \rightarrow S2 \rightarrow S3 \rightarrow S5$. Note that expired nodes and edges are never deleted from the flow graph, enabling Sphinx to accurately determine the updated current path during reroutes. This allows for the possibility that a reroute might not result in the issuance of fresh FLOW_MOD commands to all the switches on the new current path, as is the case during the reroute at t_3 (where switches $S1$ and $S2$ receive fresh instructions from the controller while $S3$ does not).

Flow graphs exploit the predictability and pattern in both topological and data plane forwarding inferred from control messages to detect attacks originating within the SDN. While flow graphs are an effective tool to verify normal and predictable network operations, they are limited in their capabilities by the nature of messages sent over the control plane and the dynamism in the topology. If there is a majority of tampered or untrusted messages, then flow graphs will perceive incorrect messages as normal behavior and not raise any alarms. Further, if the network topology changes very frequently, then several of the learned invariants may be violated, resulting in alarms.

C. High-level approach

Key Idea. Sphinx gleans topological and forwarding state metadata from OpenFlow control messages to build incremental flow graphs and verify all SDN state in realtime, including detection of security attacks on topology and data plane forwarding (such as those listed in § III-A and later in § VIII) or violations of administrative policies. Any deviant behavior is flagged and reported.

Figure 2 shows Sphinx's workflow, which involves three stages. First, Sphinx monitors all controller communication and identifies relevant OpenFlow messages required to build a comprehensive view of the network. Second, Sphinx analyzes these OpenFlow messages and extracts topological and forwarding state metadata to incrementally build a network graph complete with traffic flows. Specifically, Sphinx maintains topological and forwarding state metadata captured from (i) incoming OpenFlow packet headers, (ii) outgoing flow path setup directives, and (iii) actual flow traffic measurements over the network links, respectively. Third, Sphinx verifies the flow's current metadata against (i) a set of permissible values of metadata gathered over the lifetime of a flow, and (ii) administrative policies. Sphinx flags known attacks using administrator-specified policies, while it leverages flow-specific behavior acquired over time to detect unforeseen and potentially malicious activity.

Sphinx does not raise alerts when it *discovers* new flow behavior. Instead, Sphinx raises alerts when it detects untrusted entities triggering *changes* to existing flow behavior, or the flow violates any administrator-specified security policy. For example, Sphinx does not raise alarms when a switch learns its neighbors. However, if any of the neighbors change on any switch port, Sphinx will immediately flag the incident since it alters the network topology and subsequently the flow graph. Additionally, Sphinx will not raise alerts on flow reroutes since they are triggered by FLOW_MOD messages from the trusted controller. This significantly lowers alarms that may be generated if detection of every new behavior is flagged, which is possible in evolving networks. Such suppression of alerts also implies that any malicious activity that precedes genuine flow behavior will be treated by SPHINX as *discovered* behavior, and will thus evade immediate detection. However, the malicious activity will be detected retrospectively when SPHINX later flags the genuine behavior as suspicious, only to be negated by the administrator.

EXAMPLE. SPHINX detects the fake topology attack as described in § III-A by extracting metadata from OpenFlow control messages to maintain a view of the topology with all the active ports per switch. SPHINX observes the FEATURES_REPLY OpenFlow message to detect controller-switch connections and port details per switch. SPHINX intercepts PACKET_IN messages that contain LLDP payload, and extracts metadata to identify valid links between switches in the flow graph. It then validates the extracted metadata against a set of acknowledged invariants, such as (i) only a single neighbor is permitted per active port at a switch , and (ii) links should be bidirectional. This host-switch-port mapping enables SPHINX to detect fake edges (from a single compromised switch or host) at the instant malicious PACKET_IN messages are received by the controller.

However, two or more colluding switches or end hosts may still poison the controller's view by creating a fake bidirectional link, thereby possibly altering the shortest routing path between other hosts on the network. SPHINX can detect such fake links by verifying data plane forwarding metadata per-flow, which captures the flow patterns of the actual network traffic along a path in the flow graph. Specifically, SPHINX uses a custom algorithm to monitor the per-flow byte statistics (by intercepting STATS_REPLY messages) at each switch in the flow path, and determines if the switches are reporting inconsistent values of bytes transmitted.

D. *Why* SPHINX *works?*

SDNs provide three key features that enable SPHINX to precisely detect security threats in realtime.

(a) Ease of analysis: SDNs are less dynamic than the Internet, and OpenFlow is much simpler than traditional communication protocols. All intelligence is centralized in the controller, where the stream of all network updates is observable. This significantly eases analysis of control messages within SDNs.

(b) Action attribution: Action attribution in SDNs is much easier than in traditional networks because of the centralized controller that has global visibility and the large amount of statistics available at the controller.

(c) Domain knowledge: If we do not consider SDNs to be a black box, then we can leverage domain knowledge about OpenFlow to develop a small, yet expressive, feature set that captures the essence of all network communication. This helps to *easily* detect changes in patterns of control messages.

V. SPHINX: DESIGN

SPHINX aims to provide accurate and realtime verification of network behavior by providing three key features. First, it monitors all relevant OpenFlow control messages originating from the switches or the controller. Second, it leverages a succinct feature set that enables efficient verification of these messages. Third, it uses a custom algorithm for fast validation of network updates as they are processed by the controller.

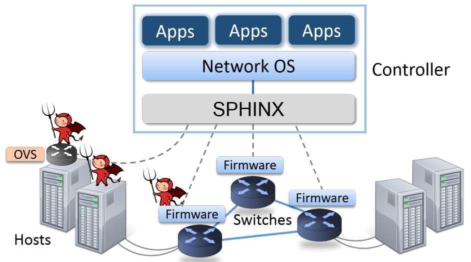


Fig. 3: SPHINX architecture.

Src MAC/IP/port	Dst MAC/IP/port	Switch and in/out-port	Flow match and statistics

Table 1: Feature set used to determine per-flow metadata.

A. *Intercept OpenFlow packets*

SPHINX must intercept every network update to be able to detect deviant behavior. Figure 3 presents a schematic architecture of the system, which shows SPHINX as a shim between the switches and the vanilla controller. An adversary, i.e., an end host or a switch, can misuse only a subset of all OpenFlow messages to poison the controller's view of the network. These messages include PACKET_IN, STATS_REPLY and FEATURES_REPLY. In contrast, the trusted controller only uses FLOW_MOD messages to direct the switches to establish connectivity between the endpoints. Thus, SPHINX actively monitors just these four OpenFlow messages to extract relevant metadata. All other messages are simply relayed through.

B. *Build incremental flow graphs*

SPHINX analyzes the OpenFlow control messages mentioned above to incrementally build and update flow graphs corresponding to each flow in the network. It then detects attacks or violations in security policies by identifying tangible changes in the network's topological and/or data plane forwarding metadata associated with every flow graph.

There are three main entities in an SDN environment that accurately characterize such metadata for each flow in the network, i.e., end hosts, switches and flows. SPHINX extracts and remembers the metadata associated with each entity to populate a feature set described in Table 1. The source/destination IP/MAC bindings provide a mapping for each host on the network. The MAC/port bindings uniquely identify a flow between endpoints. The flow match along with switch in- and out-port determines the set of waypoints for a flow in the data plane. Lastly, flow statistics provide bytes/packets transferred for every flow. Additionally, SPHINX assimilates and remembers this flow-specific physical and logical topological bindings for the end points, and the forwarding state specified by FLOW_MOD messages at each intermediate switch in the flow path, to detect potentially malicious metadata updates.

SPHINX relies on four OpenFlow messages—FLOW_MOD, PACKET_IN, STATS_REPLY and FEATURES_REPLY—to extract all relevant metadata as observed at a particular switch and port. Specifically, SPHINX determines onset of flows and topological information (host IP-MAC, MAC-port or switch-port bindings) when switches issue a PACKET_IN. The desired paths to be taken by flows, and any subsequent updates, are determined when the controller issues a FLOW_MOD for the switches. SPHINX uses

Feature	Description
Subject	(SrcID, DstID), where ∀ SrcID and DstID ∈ {Controller | WaypointID | HostID | $*$}
Object	{Counters | Throughput | Out-ports | Packets | Bytes | Rate | Match | Waypoint(s) | Host(s) | Link(s) | Port(s) | etc.}
Operation	In | Unique | Bool (True, False) | Compare ($\leq, \geq, =, \neq$) | etc.
Trigger	Packet_in | Flow_mod | Periodic

Table 2: Sphinx's policy language.

STATS_REPLY, which is received periodically from the switches, to extract flow-level statistics in the data plane, including packets/bytes transferred. Sphinx intercepts FEATURES_REPLY to glean switch configuration, including port status, when a switch first connects to the controller.

C. Validate network behavior

Verification of constraints on network entities, resources and flow properties is performed by Sphinx's policy engine. In most cases, Sphinx can quickly verify the diverse effects of all network updates on individual flows by simply traversing the flow graph and inspecting the associated metadata for conformance with application- or administrator-specified safety properties. However, processing the entire flow graph on each network update is time consuming. Thus, Sphinx caches the waypoints of the current path to determine if the update satisfies the constraints or not. In case the network updates modify the structure of the current path, such as VM migrations in multi-tenant data centers, Sphinx discards the cached waypoints, rebuilds the current path and traverses it to check for consistency (such as waypoint dependencies, etc.), and any administrator-specified security policies.

Incremental flow graphs along with the flow metadata ensure that the validation process is quick, since at each update Sphinx only has to reason about the metadata concerning a specific network link for a single flow. This design not only makes constraint verification extremely fast, but also makes action attribution easier and precise.

We next describe Sphinx's policy engine and its role in the validation of network behavior in greater detail.

VI. Sphinx policy engine

A. Constraint specification

Sphinx validates all flow graphs against a set of constraints. These constraints are of two types—(i) any administrator-specified security policies, and (ii) those acquired over time for a specific flow. Administrator-specified policies defend against *known* attacks or violations, while constraints assimilated over time can detect even unanticipated and harmful network updates.

Sphinx provides a light-weight policy framework that enables administrators to specify validation checks on incremental flow graphs. These administrator-specified constraints must be expressed in a policy language as specified in Table 2. Most modern controllers allow applications and modules to implement separate checks making policy enforcement buggy and hard. In contrast, Sphinx provides a pluggable framework to enforce complex security checks at one central location. Note that Sphinx assumes logical correctness of the policies. Validation of policies is out of scope of the current work, and is left for future work.

```
<Policy PolicyId="Waypoints">
    <Subjects><Subject value="H3, *" /></Subjects>
    <Objects>
        <Object><Waypoint value="S2" /></Object>
        <Object><Waypoint value="S3" /></Object>
    </Objects>
    <Operation value="IN" />
    <Trigger value="Periodic" />
</Policy>
```

Fig. 4: Example policy to check if all flows from host $H3$ pass through specified waypoints $S2$ and $S3$.

Each policy has four main components—subject, object, operation and trigger. The *subject* identifies traffic flow(s) between a source/destination pair in either the control or data plane (where either or both can be wildcards) over which constraints are expressed. An *object* is a keyword that specifies a traffic property describing the nature of constraints, while the *operation* specifies a relation describing the approved values that the object can attain for the given traffic flow(s), as specified by the subject. Lastly, the policy must also specify a *trigger* instructing Sphinx when to schedule the check.

Sphinx feeds the policy to a verifier, which ensures that the constraints are checked at the specified trigger. For each policy, the verifier extracts the flow and the associated flow properties, and invokes a built-in checker to evaluate the constraint. Sphinx provides several built-in checkers, including those for enforcement of policies listed in Table 4. Figure 4 shows an example policy to check if all flows originating at a host $H3$ in the network pass through specified waypoints, such as a firewall. The policy applies to all destinations in the network, as indicated by '$*$' in the 'DstID' field of the subject. The objects define the set of waypoints, while the operation 'IN' directs the verifier to check the waypoints for membership within the objects specified by the policy. The policy is checked 'periodically' as specified by the trigger.

Apart from validating the administrator-specified constraints, Sphinx automatically generates flow-specific constraints by observing updates to flow-specific topological and forwarding states, i.e., IP-MAC or switch-port bindings, forwarding actions at specific waypoints, etc., over time. These topological and forwarding states are the default constraints for that flow, and Sphinx checks for any atypical flow patterns by identifying changes to the flow's metadata. Sphinx raises an alarm if any of these invariants are violated during the duration of the flow. For example, if Sphinx receives flow-level statistics from a switch not on the flow's current path, it raises an alarm because an intermediate switch on the current path could be siphoning off flow traffic.

SDN controllers utilize graph theoretic algorithms to ensure that the computed path between a pair of endpoints observes certain standard properties, such as reachability, the absence of loops or blackholes, etc. Since Sphinx trusts the controller, the policy language currently does not allow specification of constraints over the flow graph structure. However it can easily be extended to do so, thereby enabling administrators to express policies to verify flow graph properties, such as loops, blackholes, reachability, etc.

B. Constraint verification

Algorithm 1 briefly describes the verification process. For each untrusted OpenFlow message (PACKET_IN and STATS_REPLY) in the packet stream, Sphinx together determines three classes

```
Input: $\mathbb{S}$ : Stream of incoming OpenFlow packets.
Output: DataStore : Data store for saving valid metadata for each flow.
  function VERIFIER($\mathbb{S}$)
    Initialize:
      $\mathbb{O}$ := Allow /*Processing of packet by default*/
      DataStore := $\emptyset$
    for all $\rho \in \mathbb{S}$ do
      MD := GET_PACKET_METADATA($\rho$)
      F := GET_FLOW_METADATA(MD)
      FG := GET_PATH_METADATA(F)
      /*Get policy and other constraints for packet*/
      $\Phi$ := GET_CONSTRAINTS($\rho$, MD, F, FG)
      /*Validate packet/path/flow metadata for $\rho$*/
      $\mathbb{O}$ := $\mathbb{O}$ $\bigwedge$ VALIDATE_PACKET(MD, $\Phi$) $\bigwedge$ VALIDATE_PATH(FG, $\Phi$)
            $\bigwedge$ VALIDATE_FLOW(F, $\Phi$)
      if (DENY == $\mathbb{O}$) then
        /*Raise alert for administrator*/
        if (/*Administrator allows alert*/) then
          /*Save all metadata in data store*/
          DataStore := DataStore $\bigcup$ SAVE_METADATA($\rho$, MD, F, FG)
        else
          /*Break from loop and stop the packet flow*/
    return DataStore
```

Algorithm 1: Verification of each incoming packet for each flow.

Metadata	Verification Purpose	Invariants
PACKET	Packet spoofing Controller DoS	MAC-IP-Switch-Port PACKET_IN rate, etc.
PATH	Flow graph consistency	Routing rules. path waypoints
FLOW	Switch DoS Flow statistics	Flow counters, Tx/Rx bytes, switch/out-port

Table 3: Example of some invariants verified by SPHINX.

of metadata—packet, path and flow—and verifies them against the set of both learnt and administrator-specified constraints. Packet-level metadata pertains to all metadata that are specific to just one specific PACKET_IN, such as information about a host's IP/MAC binding, or link connection between two switches. Path-level metadata refers to all metadata that describe the network's actual forwarding state behavior, such as the switch and port from which the packet was received. Note that both packet- and path-level metadata, describing the logical and physical topology and the flow paths, are obtained exclusively from PACKET_IN messages. Flow-level metadata quantify the actual data plane forwarding in the network, and are extracted from the STATS_REPLY messages received periodically.

The aforementioned metadata verification is either deterministic or probabilistic. Topological state verification can proceed even before the actual traffic has begun, i.e., it verifies properties involved in setup of flow paths and is deterministic. Verification of data plane forwarding state requires a flow to be setup, and probabilistically verifies properties that quantify the nature of the flow. Table 3 lists the three metadata classes and some of the corresponding invariants observed during verification. Table 4 lists the default policies that SPHINX checks at each verification trigger. Note that SPHINX does not verify the trusted FLOW_MOD messages. However, the effects of these FLOW_MOD messages may violate some administrator-specified policy, e.g., all flows must pass through a firewall. Thus, SPHINX validates such policies on the specified trigger.

1) Topological state constraint verification: Topological constraints, i.e., both network invariants as well as administrator-specified, can be verified using the metadata gleaned from the received PACKET_IN. Once the default invariants have been verified, the metadata are compared against all applicable policies, and any deviant behavior is flagged.

Trigger	Policy
PACKET_IN	IP-MAC binding is permissible. Network topology (physical/logical) change is permissible.
FLOW_MOD	–
Periodic	Throughput for a flow/switch port is below a threshold. Switch must not drop or siphon off packets in the flow.

Table 4: Default policies checked by SPHINX on every trigger.

Examples of such packet-level metadata verification include the detection of packet spoofing for both logical and physical topological tampering. All such verification is deterministic and fast due to incremental flow graphs, which allows verification to proceed over the last edge or metadata that was added to the graph. This also enables precise action attribution.

2) Forwarding state constraint verification: Verification of forwarding constraints in the data plane requires the validation of both packet- and flow-level metadata, which may be either deterministic or probabilistic depending on the nature of constraints involved. For example, if malicious switch(es) tamper with existing flows, then such inconsistencies may not be reflected in the analysis of flow graph structure alone. Such cases may only be determined by using flow consistency checks. Thus, SPHINX performs additional periodic checks on the flow graphs and the associated metadata to determine conformance with flow dependencies and constraints, like detecting if a flow's throughput is within a threshold, packet drops or siphoning due to malicious switch(es), etc.

OpenFlow's asynchronous nature may cause messages to arrive in an out-of-order manner at the controller. While packet-level metadata (e.g., rate of PACKET_IN messages) remains unaffected, a key challenge for SPHINX is to accurately determine flow-level statistics in the presence of unsynchronized messages from multiple different switches in the flow path, which may report flow-level statistics at different time granularity. SPHINX overcomes the above challenge using a custom algorithm that relies on an honest majority of switches along a flow path to approximate the byte and packet statistics at the flow-level. Since undesirable behavior by a malicious (or misconfigured) switch may manifest itself in traffic flowing across the switches, SPHINX generates a metric called **Similarity Index** (**Σ**) at each switch to represent the nature of the traffic flow. The Σ of a switch at timestep t is calculated as: $\Sigma_t = \Sigma_{t-1} + (\Delta_n - \Delta_{n-p})/p$, where $\Delta_n = s_n - s_{n-1}$, and s_n represents the latest (n^{th}) byte-level statistics available at timestep t. Σ is thus calculated as a moving average of the difference in byte-level statistics reported for each flow per switch in the current flow path. SPHINX chooses the last $p = 4$ statistics reported by STATS_REPLY messages, which span a few seconds and are controller dependent. This interval is sufficient to even out traffic bursts, congestion at waypoints and account for out-of-order messages, thereby avoiding false alarms. Σ also enables SPHINX to check for the presence of malicious switches that may add/drop packets at coarse timescales (at most equal to the frequency of STATS_REPLY messages).

For a particular flow, Σ must be similar for honest switches on its path till the flow encounters a malicious (or misconfigured) switch, which may inject or siphon off traffic. However, it is still possible that the malicious switch fakes the statistics with Σ similar to honest switches. Even in this case, the switches downstream would report higher (or lower) Σ if the switch is injecting or siphoning off traffic. Since offending

```
Input: $F$ : Flow, $\tau$ : threshold
Output: $\mathbb{O}$ : $\{S\}$ Set of contentious switches along the flow $F$
  function FLOW_CONSISTENCY_ VALIDATOR($F$, $\tau$)
    Initialize:
      $FG$ := GET_FLOWGRAPH($F$) : The complete flow graph for flow $F$
      $CurrP$ := GET_CURRENTPATH($FG$) : The active current flow path for $FG$
      $\Sigma_{avg}$ := 0 : Initialize running average of Similarity Index for $FG$
      $\mathbb{O} := \emptyset$
    /*Validate byte consistency for switches on $CurrP$*/
    for all $S \in CurrP$ do
      $M$ := GET_METADATA($S$)
      $\Sigma$ := SIMILARITY_INDEX($M$)
      /*Check if $\Sigma$ is an outlier*/
      if FALSE == CHECK_VIOLATION($\Sigma_{avg}, \Sigma, \tau$) then
        $\Sigma_{avg}$ := UPDATE_RUNNING_AVERAGE_INDEX($\Sigma_{avg}, \Sigma$)
      else  $\mathbb{O} \cup = \{S\}$ /*Add S to output set*/
    /*Validate inactivity of switches not on $CurrP$*/
    for all $S \in FG \wedge S \notin CurrP$ do
      $M$ := GET_METADATA($S$)
      $T$ := GET_THROUGHPUT($M$)
      /*Check if switch $S$ is not inactive*/
      if $T != 0$ then  $\mathbb{O} \cup = \{S\}$ /*Add S to output set*/
    return $\mathbb{O}$
```

Algorithm 2: Checking byte consistency across a flow.

switches cannot fake their identity (as switches connect with the controller over separate TCP connections), they would thus be pinpointed. Note that Σ will not change if malicious switch(es) compromise the integrity of the flow packets, or inject and remove an equal amount of packets from the flow traffic. To prevent such attacks on integrity of flow traffic, SDNs can leverage cryptographic mechanisms.

Algorithm 2 describes the steps to perform byte consistency checks for a given flow graph. The algorithm takes as input a flow graph and computes the current path for the flow. It then iterates over all switches in the current path to access the byte and packet statistics, and calculates the Σ for each switch. The algorithm reports a violation if it determines that a switch in the flow path reports Σ much different from the moving average Σ for the flow. The algorithm also checks for inactivity of all switches not in the current path. This verifies that no switch off the current flow path is injecting or siphoning off traffic. Further, the algorithm takes as input a threshold (τ), which is a margin of similarity used to perform outlier detection. A $\tau = x$ means that Σ at each switch along the flow path must lie between Σ/x and $\Sigma * x$. Lesser τ means lesser variability in Σ, implying stricter consistency checks. However, a lesser τ may lead to false alarms, whereas a higher τ may lead to lack of genuine alarms. $\tau = 1$ allows no margin for variability in Σ.

SIMILARITY INDEX, LINK LOSS AND τ. If two adjacent switches S_n and S_{n+1} share a link with loss rate ρ, and the average similarity index for the flow path till S_n is Σ_{avg}, then Σ_{n+1} for the next switch in the flow, i.e., S_{n+1}, will be proportional to the loss rate: $\Sigma_{n+1} \propto \Sigma_{avg} * (1 - \rho)$. SPHINX raises an alarm if Σ_{n+1} is not within the threshold τ. In other words, SPHINX will not raise an alarm if the following holds true: $1/\tau < (\Sigma_{n+1} \ / \ \Sigma_{avg}) < \tau$. Solving the above equations, we get $\tau \leq k/(1 - \rho)$, where k is the proportionality constant.

C. Handling alarms

If a violation is detected during verification, SPHINX raises an alarm for vetting by the administrator. SPHINX also automatically generates reports that pinpoint the cause of the alarm. For deterministic verification, SPHINX lists the offending packet and the link/waypoint responsible for the alarm. For probabilistically verified invariants, SPHINX gives the exact switch/out-port along the flow where the validation failed. Once the alarm is vetted by the administrator, SPHINX learns the new behavior and incorporates it in its metadata store, preventing further alerts. If the administrator marks a flow as suspicious, the metadata for that run is discarded.

Attack	ODL	Floodlight	POX	Maestro
ARP poisoning	✓	✓	✓	✓
Fake topology	✓	✓	✗	✓
Controller DoS	✓	✗	✓	✓
Network DoS	✓	✓	✓	✓
TCAM exhaustion	✗	✓	✓	✓
Switch blackhole	✓	✓	✓	✓

Table 5: Comparison of controller vulnerability.

VII. IMPLEMENTATION

We envision SPHINX to be integrated within the SDN controller as a module/application. However, to demonstrate SPHINX's broad utility and compatibility with different controllers, and also for ease of implementation, we implement it as a controller-agnostic proxy that sits between the controller and the switches. SPHINX is written in ~2100 lines of JAVA, and leverages the Netty I/O library [11]. It implements separate queues for switch to controller communication (PACKET_IN, FEATURES_REPLY and STATS_REPLY) and controller to switch communication (FLOW_MOD), for enhanced performance.

SPHINX is compatible with OpenFlow v1.1.0, and works with both OpenDaylight v0.1.0 (ODL) and Floodlight v0.90 controllers. SPHINX can easily be integrated with other controllers such as Maestro and POX, and requires no significant changes. However, we needed to modify just 30 lines in ODL to ensure conformance with our design. Specifically, ODL installs flow rules based on destination IP only. Since SPHINX defines flow rules as a MAC-MAC address pair of the endpoints, we modified ODL to output source and destination MAC addresses in the FLOW_MOD messages.

SPHINX may also be implemented as a passive monitoring tool that replicates all control traffic at the switches and analyzes them separately. This is feasible as all switches are equipped with port mirroring. However, since the switches are untrusted and port mirroring provides no reliability, i.e., it may drop traffic, we did not implement this mechanism.

VIII. STUDY OF CONTROLLER VULNERABILITIES

We now describe empirical studies to demonstrate the relative ease of launching attacks against four commonly used SDN controllers—ODL, Floodlight, POX and Maestro. We also describe how SPHINX successfully detects each of the attacks. While, some of the attacks were detected using administrator-specified policies, others were automatically detected by SPHINX using flow-specific permissible behavior assimilated over time. Table 5 lists the results of our experiments. It indicates that popular controllers are vulnerable and can be easily exploited. The vulnerabilities described here afflict SDNs in general and are not specific to a particular controller.

A. Attacks on Network Topology

1) ARP Poisoning: Compromised hosts can spoof physical hosts by forging ARP requests, i.e., ARP poisoning, fooling the controller into installing malicious flow rules to divert

```
<Policy PolicyId="ARP-poisoning">
  <Subject value="H5, *" />
  <Object><Host value="IP, MAC" /></Object>
  <Operation value="9.12.34.56, 60:67:20:f1:b7:4c" />
  <Trigger value="PACKET_IN" />
</Policy>
```

Fig. 5: Example policy to detect ARP poisoning by validating host $H5$'s IP/MAC bindings.

traffic flows, possibly for eavesdropping, thereby allowing a malicious host to intercept traffic intended for another host. Malicious hosts along with an accomplice can also initiate arbitrary flows to fool the switch and the controller into installing flow rules that create loops or blackholes in the network or mount an IP splicing attack. We implement the attack using a topology of three hosts connected to a switch—a malicious host A, and two benign hosts B and C. The attack involves sending spoofed ARP requests 'Who has B, tell C' but with A's MAC address. These malicious ARP requests are relayed as PACKET_IN messages to the controller, and ultimately corrupt B's ARP cache along with the controller's view of the topology, which then routes traffic from B (intended for C) to A instead. We test the attack by sending repeated PING requests to B from C. Instead of observing the responses at C, we observed the responses at A. Note that variants of this attack are possible with *any* packet triggering a PACKET_IN message, and not just the ARP packet. This attack works across all the controllers we tested. Our video demo shows a variant of this attack for ODL [1].

DETECTION. SPHINX builds a flow graph that maintains and updates MAC-IP bindings for all hosts in the network along with a list of possible switch-ports they can be located at. It extracts this metadata when a PACKET_IN arrives. If any deviation from these permissible bindings is observed during a PACKET_IN, SPHINX flags it and raises an alarm. In case the administrator permits a flagged binding, SPHINX updates its list accordingly to prevent further alarms. ARP poisoning can also be detected using custom policies written using SPHINX's policy language. Figure 5 shows an example policy that raises alarms if SPHINX detects a different binding for host $H5$ in its metadata store other than as specified by the policy.

2) Fake topology: We implement the host-based variant of the attack as described in § III-A, where a single malicious host tries to create a fake network link, using a linear topology of three switches X, Y and Z, with server A connected to switch X, and server B connected to switch Z. Server A sends a malicious LLDP packet, spoofing it to have come from switch Z. The attack creates a fake unidirectional edge from Z to X in the controller's view, which results in recomputation of routing paths. Our video demo shows a variant of this attack for ODL [6]. Following the addition of the fake edge, PING responses from B will not reach A (for the corresponding PING requests from A to B). While ODL, Floodlight and Maestro allow the creation of fake unidirectional edges, POX validates a link only if adjacency is both ways. Thus, except POX, other controllers can be tricked using a single malicious end host. For POX, an accomplice will suffice to trick the controller. Similarly, compromised soft switches can also fool the controller by sending spoofed LLDP packets.

DETECTION. As described earlier, SPHINX extracts metadata from PACKET_IN and FEATURES_REPLY messages to build a flow graph that learns and maintains a view of the topology with all the active ports per switch. These metadata are validated against invariants such as the bidirectionality of a network edge between switches, and the presence of only a single neighbor per active port at a switch. Thus, the host-switch-port invariant ensures that no fake edges are ever added to the network. LLDP spoofing can also be detected using custom policies written using SPHINX's policy language. Figure 6 shows an example policy that raises alarms if SPHINX detects different switch-port bindings for a link between switches $S1$ and $S2$ in its metadata store other than as specified by the policy.

```
<Policy PolicyId="LLDP-spoofing">
  <Subject value="S1, S2" />
  <Object><Link value="SrcPort, DstPort" /></Object>
  <Operation value="P3@S1, P5@S2" />
  <Trigger value="PACKET_IN" />
</Policy>
```

Fig. 6: Example policy to detect LLDP spoofing by checking if a link between switches $S1$ and $S2$ exists on valid ports.

```
<Policy PolicyId="Controller-DoS">
  <Subject value="*, Controller" />
  <Object><Throughput value="50" /></Object>
  <Operation value="≤" />
  <Trigger value="Periodic" />
</Policy>
```

Fig. 7: Example policy to detect controller DoS.

NOTE 1. The default flow-specific invariants provide comprehensive detection of unanticipated changes in the topological and forwarding state behavior over the entire network. In addition, the policies provide the administrator with control to specify fine-grained constraints over the flow-specific topological and forwarding state of specific network entities. Thus, the two mechanisms complement each other.

NOTE 2. While ARP poisoning and LLDP spoofing corrupt the physical topological state, fake IGMP messages from a malicious host can corrupt the logical topological state. In § IX-C, we discuss how malicious entities can spoof logical topological state and how SPHINX detects against such attacks.

B. Attacks on Data Plane Forwarding

1) Controller DoS: OpenFlow requires the switches to send complete packets to the controller if the ingress queues are full. Such control plane flooding may significantly increase the computational load on the controller and even bring it down. We tested this using Cbench [2] to flood the controller with high throughput of PACKET_IN messages for installation of new flows, thereby hampering the normal operation of the SDN controller. On increasing the number of switches and hosts in the network, all controllers except Floodlight exhibited DoS-like conditions, i.e., either the controller breaks down or the network latency increases to inordinate timescales. Unlike other controllers, Floodlight throttles incoming OpenFlow messages from the switches to prevent DoS. However, the connection of the switches with the controller snaps when a large number of switches attempt to connect with it.

DETECTION. SPHINX detects control plane DoS attacks on the SDN controller by observing flow-level metadata to compute the rate of PACKET_IN messages. SPHINX raises an alarm if this throughput is above the administrator-specified threshold. Figure 7 shows an example policy that reports violation if the PACKET_IN throughput on any link from the switches to the controller reaches 50 Mbps.

```
<Policy PolicyId="Network-DoS">
  <Subject value="*" />
  <Object><Throughput value="100" /></Object>
  <Operation value="≥" />
  <Trigger value="Periodic" />
</Policy>
```

Fig. 8: Example policy to detect network DoS.

```
<Policy PolicyId="TCAM-exhaustion">
  <Subject value="Controller, S5" />
  <Object><Rate value="50" /></Object>
  <Operation value="≤" />
  <Trigger value="FLOW_MOD" />
</Policy>
```

Fig. 9: Example policy to detect TCAM exhaustion.

2) Network DoS: We tested the four controllers for network DoS by installing custom rules on two OVSes in our topology, to direct traffic into a loop and thereby magnify a 1 Mbps flow between a specified endpoints such that it completely chokes a 1 Gbps link. An *iperf* session between arbitrary hosts across the choked link yielded a bandwidth of just ~400 Kbps. We also observed that the attack completes in sub-second time intervals for all the controllers.

DETECTION. For every flow, SPHINX periodically updates the flow graph with byte statistics reported by the switches across the flow path, and validates this byte consistency with the intended behavior by monitoring FLOW_MOD messages. Figure 8 shows an example policy to detect if the throughput across any network link rises above the administrator-specified threshold of 100 Mbps. Additionally, SPHINX leverages path- and flow-level metadata to detect loop formation in the network.

3) TCAM exhaustion: We test the controllers for TCAM exhaustion attack as described in § III-A using a switch (IBM RackSwitch G8264 with a TCAM of size $1K$) with three hosts (A, B and C). We repeatedly send exactly $1K$ flows from host B, with arbitrary source addresses, to ensure that flow rules never time out at the switch. Thus, any new flow rule (say those corresponding to PINGs from A to C) are not installed, thereby causing a denial of service. The TCAM exhaustion attack worked for Floodlight, POX and Maestro, which completely populate the TCAM (as they use source/destination IP pairs as keys). This causes them to exhibit high latencies (40-80 ms) for any new flow rule installation (even PINGs), which creates near DoS conditions for normal network operations. In contrast, the attack did not work with the vanilla ODL controller, since it installs rules only using the destination IP as the key. In our experiment, since we sent all traffic to a single destination, only a single rule was installed for all $1K$ flows. To exhaust the TCAM in an ODL setup, we need flows with unique destination IPs that are within the subnet.

DETECTION. SPHINX populates the flow graph with packet-level metadata for FLOW_MOD messages to compute the rate of flow installations. SPHINX detects TCAM exhaustion if this rate continues to be high over time and violates administrator-specified policy directives, as shown in Figure 9. The example policy raises a violation if the FLOW_MOD throughput from the controller to switch $S5$ is greater than 50 FLOW_MOD messages per second.

4) Switch blackhole: A blackhole is a network condition where the flow path ends abruptly and the traffic cannot be routed to the destination. SPHINX trusts the controller, which ensures that blackholes are not formed at the instant flow paths are setup [2]. However, a malicious switch in the flow path may drop or siphon off packets, thereby preventing the flow from reaching the destination. We tested the four controllers for the above variant of the switch blackhole attack in a flow path of 5 switches by installing custom rules on one of the OVSes (not including the ingress and egress switches) to drop all packets.

DETECTION. SPHINX determines the switch blackhole attack associated with switches by verifying the flow graph for byte consistency, which captures the flow patterns of the actual network traffic along a path in the flow graph. Specifically, SPHINX uses Algorithm 2 to monitor the per-flow byte statistics at each switch in the flow path, and determine if the switches are reporting inconsistent values of bytes transmitted than expected. If the bytes reported across the switches fall below a threshold, SPHINX raises an alarm. In this case, the blackhole causing switch causes the successor switch in the flow path to report 0 bytes for the corresponding flow, thereby triggering the alarm.

IX. EVALUATION

We now present an evaluation of SPHINX. In § IX-A, we evaluate SPHINX's accuracy by measuring how quickly it can detect attacks, the effectiveness of the byte consistency algorithm, and the false alarms generated under benign conditions. In § IX-B, we measure user perceived latencies introduced by SPHINX, variation in packet throughputs, overhead of policy verification, etc., and also compare its performance against related work. Lastly, in § IX-C, we describe our experiences with SPHINX under four diverse case studies.

EXPERIMENTAL SETUP. Our physical testbed consists of 10 servers connected to 14 switches (IBM RackSwitch G8264) arranged in a three-tiered design with 8 edge, 4 aggregate, and 2 core switches. All of our servers are IBM x3650 M3 machines having 2 Intel Xeon x5675 CPUs with 6 cores each (12 cores in total) at 3.07 GHz, and 128 GB of RAM, running 64 bit Ubuntu Linux v12.04.

We determine the default value of τ in SPHINX empirically. The proportionality constant k (recall § VI-B2) for our physical testbed was empirically determined to be 1.034, and for link loss rates of up to ~1%, the default τ comes out to be 1.045. Thus, Σ at each of the switches along the flow path in our testbed must lie between $\Sigma/1.045$ and $\Sigma * 1.045$.

TOOLS USED. We use several tools for evaluating SPHINX in a controlled setup. We achieve scalability using the Mininet emulator with the number of hosts varying from 100 to $10K$. We use Cbench [2] to stress test SPHINX's performance in the presence of a large number of hosts with high PACKET_IN rates. Cbench emulates switches and hosts to stress the controller with PACKET_IN messages that generate FLOW_MOD rules to be installed on switches. We use the Mausezahn packet generator [9] to control the rate of TCP packets from several Mininet hosts to stress SPHINX with varying FLOW_MOD rates. We use tcpreplay [18] to vary PACKET_IN rates. Lastly, we use custom scripts to generate benign traffic in Mininet.

[2] A static blackhole could manifest if the 'action' attribute of the OpenFlow FLOW_MOD message received at a switch may not have any associated out-port, or the 'action' might send the packet back on the received port itself. Thus, the switch will either drop all packets, or return them along the in-port.

Attack	Detection time (μs)	
	Physical testbed	$1K$ Mininet hosts
ARP poisoning	44	60
Fake topology	66	80
Controller DoS	75	900
Network DoS	75	164
TCAM exhaustion	n/a	n/a
Switch blackhole	75	900

Table 6: Attack detection times (μs) using SPHINX. Controller DoS was performed with ODL as Floodlight throttles high packet rates.

A. Accuracy

1) Attack detection: We measure SPHINX's detection accuracy under two different parameters. First, SPHINX must provide near realtime detection of attacks. Second, even in the presence of diverse network traffic and multiple different faults, SPHINX should be able to quickly detect each attack.

For the first experiment, we introduced synthetic faults (described in § VIII) along with benign traffic on our physical testbed and with $1K$ emulated hosts in Mininet (arranged in a tree topology with fanout 10 and depth 3). We then used SPHINX to measure the absolute time taken to detect the faults. We define detection time as time taken to raise an alarm from the instant SPHINX received the offending packet. We used a custom traffic generator to introduce benign traffic with 300 FLOW_MOD/sec. We repeated each scenario 10 times and report the results in Table 6. The results show sub-millisecond detection times, which indicates that SPHINX provides near realtime detection of attacks, even with $1K$ hosts and reasonable background traffic. Note that ARP and fake topology attacks are detected when PACKET_IN messages are processed. However, SPHINX runs a periodic flow graph validator to detect DoS attacks. Thus, these detection times may vary as size of the flow graph increases.

For the second experiment, we used Mininet to scale the number of hosts from 100, $1K$, up to $10K$. We then launched ARP poisoning, fake topology and network DoS attacks simultaneously in different parts of the network. We repeated each experiment 10 times, and observed that SPHINX successfully detected all the faults under the different topologies.

BENIGN TRAFFIC. We sanity check SPHINX's deterministic verification by measuring the false alarms generated in the presence of benign traffic with all the checks in Table 4 enforced. We wrote a traffic generator that uses three diverse real-world, but benign, network traces—a 14min trace from LBNL [7], a 65min trace [4], and a 2hr trace extracted from [3]—to drive traffic in Mininet. Execution of these traces raised no alarms at the default τ of 1.045.

DIAGNOSTICS. SPHINX provides useful diagnostic messages to pinpoint the real cause of attacks. SPHINX can do so because it (i) succinctly captures the flow metadata, and (ii) wherever possible, maps each network update to an incoming OpenFlow packet. For example, in the fake topology attack, SPHINX provides diagnostic messages to identify the malicious LLDP packet, and also lists the in- and out-port of the source and destination switches to identify the network link over which the offending packet was sent.

2) Sensitivity of τ: SPHINX's accuracy of probabilistic verification is influenced by τ (see § VI-B2), which may lead to false alarms or the absence of genuine alarms. We study τ's impact under two scenarios using controlled experiments. First, we measure the probability of alarms generated due to competing, but genuine flows over shared links with different values of τ. Note that these would be false alarms since the flows are genuine. Second, we study the probability of lack of genuine alarms, even in the presence of a misbehaving switch or link. Such genuine alarms should have been raised by SPHINX's verification checks, but did not because of τ.

(a) False alarms: We performed a worst-case analysis of false alarms raised for a given τ using competing TCP *iperf* flows. TCP's fair share nature will generate fluctuations in throughput to cause changes in the switches' Σ along the flow path, which would raise alarms. We used Mininet hosts that share a 3 hop path, and compute the fraction of Σ verification checks that raised false alarms. We observed that as τ increases, the probability of observing false alarms decreases (see Figure 10a). Both precision and recall are 0, since there are no true positives. At the default $\tau = 1.045$, we observed 6 alarms for 8 competing flows over 5 mins. We also performed this experiment on our physical testbed, which yielded similar results. Note that loss of STATS_REPLY messages, which provide cumulative statistics, may also lead to false alarms depending on τ.

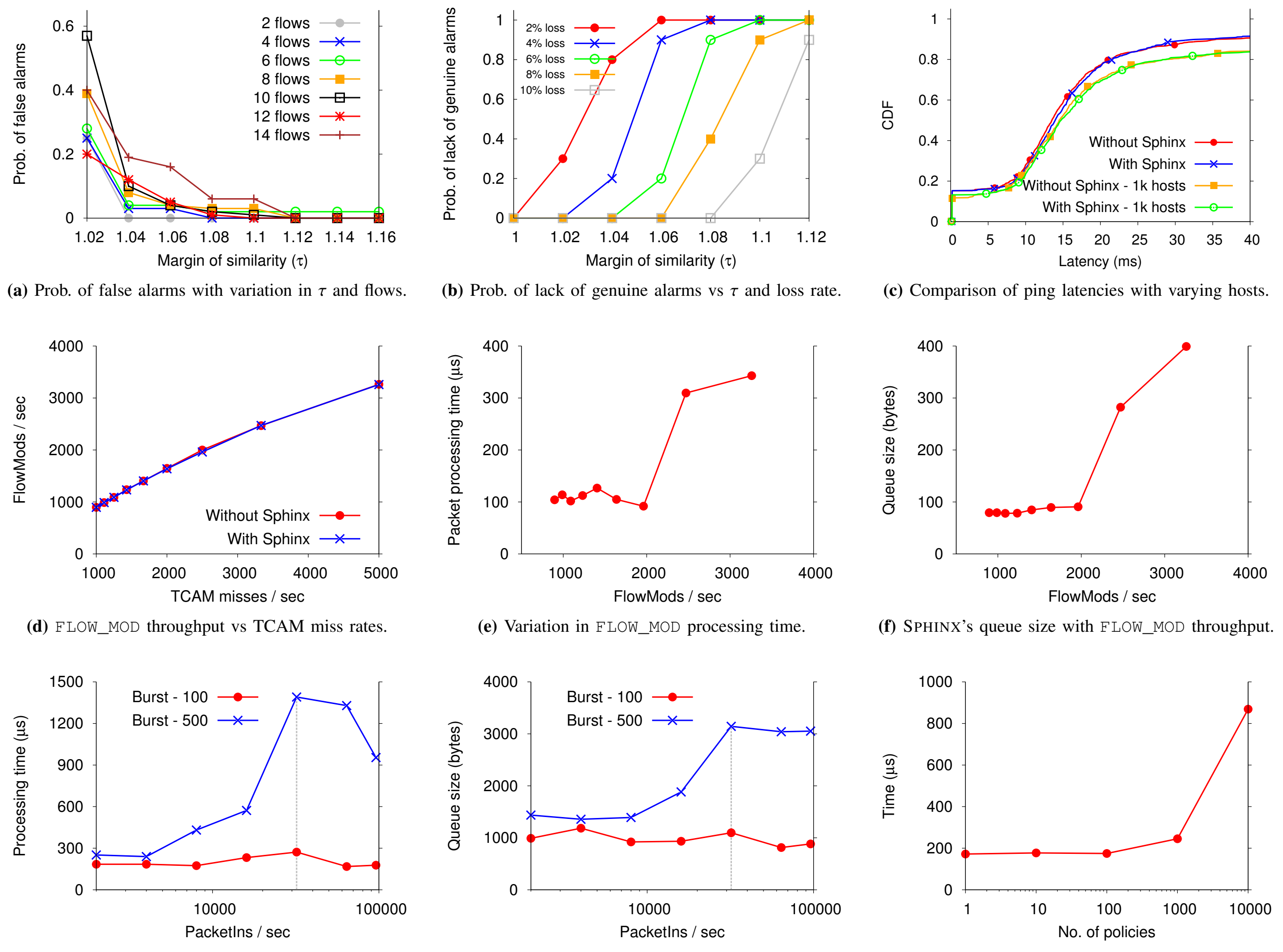
(b) Lack of genuine alarms: We define the probability of the lack of genuine alarms for a given τ as the ratio of the number of checks that did not trigger an alarm to the total checks triggered during verification. We evaluated the above metric for controlled flows between Mininet hosts that are 6 hops apart. We introduced packet drops on one link in the path to mimic a misbehaving switch or link. Alarms will be triggered because of the variability in Σ due to packet drops. However, SPHINX might suppress some of these genuine alarms. We observed that as τ increases, SPHINX underreports violations, and thus the probability of lack of genuine alarms during verification increases (see Figure 10b). For a given τ, both precision and recall are the same, i.e., equal to one minus the probability of lack of genuine alarms at each data point.

B. Performance

We perform experiments with both ODL and Floodlight. However, in the interest of space we report results with Floodlight only. All experiments check policies listed in Table 4.

1) End user latencies: We compute the overhead of using SPHINX as perceived by end users by observing RTTs for PING packets between two hosts separated by 5 hops in our physical testbed. We modified Floodlight to install rules with an idle timeout of 1 sec, and used Cbench to understand the effect of increasing number of hosts on the observed PING latencies. We send $1K$ PING packets at intervals of 3 sec, thereby causing each PING to result in a FLOW_MOD. Figure 10c shows the results of the experiment. For clarity, we only plot scenarios with 1 and $1K$ hosts. We observe that the latency increases with increasing number of hosts. However, even with $1K$ hosts, the latency overhead of SPHINX at the 50% mark is just 300μs. With $10K$ hosts, we observed much less latency for both cases with and without SPHINX. We attribute this reduced latency to Floodlight, which throttles messages at high throughput.

2) FLOW_MOD throughput: End user latency is also affected by how quickly SPHINX can process FLOW_MOD packets and



(a) Prob. of false alarms with variation in τ and flows. **(b)** Prob. of lack of genuine alarms vs τ and loss rate. **(c)** Comparison of ping latencies with varying hosts. **(d)** FLOW_MOD throughput vs TCAM miss rates. **(e)** Variation in FLOW_MOD processing time. **(f)** SPHINX's queue size with FLOW_MOD throughput. **(g)** PACKET_IN processing times. **(h)** SPHINX's queue size with PACKET_IN rates. **(i)** Policy verification times with increasing policies.

Fig. 10: SPHINX evaluation.

forward them to the switches for flow setup. We measure SPHINX's overheads in processing FLOW_MOD messages by observing FLOW_MOD throughput with increasing rate of incoming TCP connections achieved with and without SPHINX. We wrote a driver program using Mausezahn to initiate new TCP connections from several Mininet hosts. Thus, each TCP packet results in a TCAM miss, which subsequently generates a PACKET_IN and elicits a FLOW_MOD message from the controller. Figure 10d shows the results. We see that even at high TCAM miss rates (or PACKET_IN rates), SPHINX maintains a high FLOW_MOD throughput. We also observe that the throughput is only constrained by the controller's overhead, which is evident from the fact that with and without SPHINX FLOW_MOD throughputs are almost equal (with at most 2% overhead).

We also measure the FLOW_MOD processing times (Figure 10e) and SPHINX's ingress queue size (Figure 10f) from within SPHINX itself. We observe that for FLOW_MOD throughput below $2K$, the processing time is below 100μs, but it rises as the throughput increases. A similar trend is observed for SPHINX's ingress queue size, which remains small at ~110 bytes for FLOW_MOD rate of $2K$, but increases to ~400 bytes at high throughput. This is because at higher FLOW_MOD rates, the controller piggybacks several OpenFlow messages in the same TCP payload and sends them in bursts. This increases both the processing times and the queue sizes.

3) PACKET_IN processing: Attack detection times are impacted by the length of SPHINX's ingress queue and the processing of PACKET_IN messages. A large queue size negatively affects SPHINX's performance, while small packet processing times affect it positively. Thus, we observe the length of the queue of unprocessed packets in SPHINX's pipeline and the time taken to process packets as the PACKET_IN rates vary. We use tcpreplay to send bursts of packets at appropriate intervals to vary the PACKET_IN rate. For the experiment, we use burst sizes of 100 and 500 packets. Figures 10g and 10h plot the packet processing times and ingress queue sizes as the PACKET_IN rate varies. We observe that as the rate increases, both processing times and queue sizes also increase. This is because as the PACKET_IN rate increases, the switches (like the controller) also piggyback OpenFlow messages, and SPHINX receives an onslaught of packets proportional to the burst size. This results in higher processing times and queue sizes.

However, both processing rates and queue sizes show a decrease after the $32K$ mark. We attribute this to throttling in Floodlight. We further stress test SPHINX using Cbench in throughput mode with $10K$ hosts at $\sim 113.6K$ PACKET_IN messages/sec. We observe a packet processing time of $\sim 2ms$, and a mean queue size of $\sim 6KB$. This is because many more OpenFlow packets are sent piggybacked at higher burst rates.

4) Policy verification: SPHINX implements validation checks on every network update. Thus, we study the impact on the processing time of FLOW_MOD messages with increasing number of security policies. Since SPHINX works with incremental flow graphs, it results in lower validation times, which positively affects SPHINX's performance. This experiment aims to show that even simple policies, such as those in Table 4, when executed a large number of times do not introduce high overheads. Figure 10i shows the results. We observe that as the policies increase from 1 to $1K$, the validation time increases by just 73μs to 245μs. Even with $10K$ policies, SPHINX takes just 869μs to complete verification of the corresponding FLOW_MOD.

5) Resource utilization: We measured SPHINX's resource consumption using Cbench with $50K$ hosts running for 20mins, and observed a peak (relative) CPU usage of $\sim 6\%$ and memory usage of $\sim 14.5\%$. The high memory utilization is due to the processing of metadata from a large number of PACKET_IN messages.

COMPARISON WITH RELATED WORK. We now put in perspective SPHINX's performance against VeriFlow [30] and NetPlumber [28], which are most closely related to it in design. While these works address problems different from ours (e.g., they do not consider malicious entities in the network, and examine flow rules for conflicts), we present these results to put SPHINX's performance in context. All three tools report sub-millisecond mean verification time. At high FLOW_MOD throughput rates, SPHINX imposes maximum overheads of $\sim 2\%$, and is only limited by the overheads of the controller itself. In contrast, VeriFlow reports a maximum FLOW_MOD throughput overhead of 12.8%. This is because VeriFlow must traverse the entire multi-dimensional *trie* for verifying each FLOW_MOD, whereas SPHINX uses pre-built incremental flow graphs for validation that require minimal processing. No similar data was available for NetPlumber.

C. Case Studies

We now show SPHINX's broad utility by illustrating how it can support disparate networking needs without major changes.

1) Network virtualization: Open DOVE [12] is an overlay network virtualization platform for data centers that provides logically isolated multi-tenant networks with L2/L3 connectivity. Open DOVE features a scalable control plane, including address, policy, and mobility management, and a VXLAN [19] based data plane. It includes several key components—network controller or management console (oDMC), connectivity server (oDCS), gateway (oDGW) and OVS(es). Connectivity between the VMs and oDMC is handled via the OVS(es). oDMC is responsible for creating and registering overlays, while oDCS performs policy enforcement. oDGW externalizes the overlays for communication with external networks.

L2 networks are vulnerable to packet spoofing and DoS attacks. However, a MAC-over-IP mechanism for delivering L2 traffic, such as VXLAN, significantly extends this attack surface. Rogue endpoints can inject themselves into the network by (i) subscribing to multicast groups that carry broadcast traffic for VXLAN segments, and (ii) sourcing MAC-over-UDP frames to inject spurious traffic and hijack MAC addresses. Recent work [40] confirms that VXLAN is susceptible to ARP poisoning (from both overlay and tenant networks) and MAC flooding (from overlay network). SPHINX can easily secure the oDMC in Open DOVE to provide robust defenses against packet spoofing and DoS attacks in network virtualization platforms. This requires only minor changes in SPHINX to enable processing of VXLAN packets instead of OpenFlow.

2) VM Migrations: The migration of VMs from one host to another is a frequent phenomenon in clouds and data center networks. Such deployments would require SPHINX to be able to identify these migrations, so as to prevent the generation of false alarms that might arise due to purported violations in the invariants associated with the migrating VM (e.g., MAC-IP-Switch-Port bindings) when it relocates to a new host.

SPHINX can achieve this by listening for RARP messages generated by the migrating VMs, along with switch-to-controller messages caused as a result of these migrations (such as notifications of changes in the port status at the source and destination switches). Alternatively, SPHINX can also listen for control messages of the cloud administrator actuating the migrations. Once SPHINX determines that a VM has migrated, the relevant metadata would be internally updated (e.g., the Switch-Port mapped to the VM's MAC-IP), and no violations would be reported. Note that migrations themselves cannot be maliciously orchestrated from one host to another, as that would entail compromising the network administrator. Further, while a malicious network entity might attempt to fake a VM migration, it would be unable to generate a valid sequence of messages from both source and destination switches in the absence of an accomplice.

3) Load balancer: Load balancers distribute incoming client requests across a set of replicated servers to maximize throughput, minimize response time, and optimize resources. Typically, clients access the service through a single public IP address reachable via a gateway, and the load balancer rewrites the destination IP of the incoming client packets to the address of the assigned replica server. Similarly, the source IP of all outgoing response packets are also rewritten to the public IP address visible to the client. In SDNs, where load balancing is implemented as a controller module, packet routing is achieved by installing rules with write actions at the gateway—OFPAT_SET_NW_DST (for incoming request packets) and OFPAT_SET_NW_SRC (for outgoing response packets)—before forwarding. A load-balanced SDN requires no additional processing on SPHINX's end, which treats the load-balanced flows as unicast flows between the client and the assigned replica.

4) Multicast: Controller applications/modules maintain multicast groups as multicast trees. Each group has a unique multicast IP that is used by members to send/receive messages. Receivers interested in joining/leaving a particular group must send IGMP messages to the controller, which are forwarded as PACKET_IN messages for maintenance of multicast groups. Malicious hosts can forge IGMP join/leave requests to multicast groups leading to DoS for legitimate members. For example, a malicious host can repeatedly send forged IGMP leave requests on behalf of an unsuspecting host A for multicast group M. This would result in the controller accordingly modifying its multicast trees by removing A from group M, which effectively results in DoS, wherein host A can never listen to communication from M. Similarly, a malicious host B can send forged IGMP join requests to make the unsuspecting host A a member of all available multicast groups, which could lead to DDoS by choking the downlink to A.

We built a multicast module for ODL to control and manage multicast trees for multicast groups, and subsequently implemented the attacks described above on vanilla ODL. SPHINX enhanced ODL is immune to such attacks, since it verifies each IGMP `PACKET_IN` on a particular switch by leveraging its view of the topology to extract the switch-port on which the request was received. SPHINX then validates if the host is connected to the particular switch. If the validation fails, SPHINX raises an alert. SPHINX leverages `FLOW_MOD` messages to identify source-based multicast routing trees for different groups and maintains the corresponding multicast flow graphs. SPHINX also performs path consistency checks, and periodic flow consistency checks on the multicast flow graph.

X. DISCUSSION AND FUTURE WORK

LIMITATIONS. SPHINX's has a few limitations, as it can only detect tangible side-effects arising from network updates.

(1) SPHINX cannot identify a malicious *ingress* or *egress* switch in a flow path that adds/drops packets to influence the Σ. This limitation is inherent to SPHINX, since it relies on `STATS_REPLY` from untrusted switches along the flow path to generate Σ and detect flow inconsistencies. Specifically, SPHINX cannot validate the Σs reported by the *ingress* or *egress* switches in the flow path. However, SPHINX can leverage supplementary data from other standard traffic monitoring techniques such as sFlow or NetFlow to perform validation at the ingress and egress switches.

(2) SPHINX might miss some transient attacks. A major challenge in detecting flow inconsistencies arises from the granularity at which metadata statistics are updated, which spans a few seconds and is controller dependent. Fixing this limitation may require changes to the controller to report flow statistics at fine grained intervals, or require SPHINX to augment its analysis with finer granularity data from sFlow or NetFlow to achieve more precision. Alternatively, SPHINX can also be augmented by making use of network monitoring frameworks such as Planck [39] and PayLess [23] for greater accuracy in link utilization measurements.

(3) The accuracy and effectiveness of flow graphs to detect security violations as described is limited by the lack of realistic networks available to us for large scale experimentation.

(4) A high value of τ may cause SPHINX to under report violations, which can be fixed by using flow-specific τ.

(5) SPHINX cannot detect compromise in packet integrity. However, cryptographic mechanisms can fix this limitation.

FUTURE WORK. SPHINX in its present form does not consider the cases described below.

(1) Flow rule aggregation: Controller modules often aggregate flow rules to conserve switch TCAM. SPHINX, as implemented, requires installation of source/destination based rules that hamper aggregation. However, SPHINX can easily be modified to support aggregated flow rules.

(2) Mixed networks: Real enterprise deployments may have OpenFlow switches interacting seamlessly with other non-OpenFlow network entities. We plan to enhance SPHINX to detect security attacks in such mixed settings as well.

(3) Proactive OpenFlow environment: The attacks as described in § III and § VIII assume a reactive OpenFlow setup, where untrusted switches and hosts may generate malicious control traffic to elicit detrimental responses from the controller that further poison its view of the network. In a proactive OpenFlow environment, a malicious controller or applications can initiate attacks on the SDN. We leave detection of such proactive attacks for future work.

XI. RELATED WORK

Recent advances in SDN security have primarily focused on security enforcement frameworks [38], [41], [42], and realtime verification of network constraints [22], [27]–[30], [34], [37], [44]. To our knowledge, SPHINX is the first system to detect a broad class of attacks in SDNs in realtime, with a threat model that does not require trusted switches or hosts.

(1) Security enforcement: FORTNOX [38] extends the SDN controller with a live rule conflict detection engine, while FRESCO [41] provides a security application development framework to enable modular development of security monitoring and threat detection applications. Both these systems focus exclusively on threats arising from malicious applications that may result in the installation of conflicting rules. In contrast, SPHINX's threat model is different, and can detect a much broader class of attacks on SDNs.

Avant-guard [42] alters flow management at switch level to make SDN security applications more scalable and responsive to dynamic network threats. However, unlike SPHINX, it focuses mostly on DoS attacks, and requires modifications to the OpenFlow protocol. In contrast, SPHINX uses succinct metadata to detect a wide array of attacks while being controller agnostic, and requires no changes to the OpenFlow protocol.

(2) Network verification: Concurrent with our work, TopoGuard [27] is a security extension to SDN controllers that detects attacks targeted to poison the controllers' view of the network topology, by fixing security omissions in the controllers. In contrast, SPHINX unifies detection of attacks on network topology and data plane forwarding using flow graphs. However, SPHINX currently detects attacks within OpenFlow-based SDNs, while TopoGuard targets mixed networks also.

Natarajan et al. [37] present algorithms to detect conflicting rules in a virtualized OpenFlow network. Xie et al. [44] statically analyze reachability properties of networks. Anteater [34] can provably verify the network's forwarding behavior and thus determine certain classes of bugs. Like Anteater, Header Space Analysis (HSA) [29] also leverages static analysis to detect forwarding and configuration errors. In contrast, SPHINX is a dynamic system that sits closer to the actual network operations. SPHINX analyzes OpenFlow control messages in realtime to build flow graphs, and detects a broad class of threats arising from untrusted hosts and switches in SDNs.

VeriFlow [30] segregates the entire network into classes with the same forwarding behavior using a multi-dimensional prefix tree. Any network update affecting the forwarding rules and specified policies can then be verified in realtime. NetPlumber [28] uses HSA incrementally to maintain a dependency graph of update rules to enforce runtime policy checking. NetPlumber can also verify arbitrary header modifications, including rewriting and encapsulation. SPHINX is similar in spirit to both VeriFlow and NetPlumber in that it leverages packet metadata to construct and analyze the forwarding state of the network on each update. Like NetPlumber, SPHINX also provides a policy framework for expressing constraints on flows. However, both these tools verify network-wide invariants by examining the flow rules installed by the controller, and assume the data plane to be free of adversaries. In contrast, Sphinx makes no such assumptions and analyzes various switch-controller messages to ensure that the actual behavior of the network conforms to the desired behavior.

XII. CONCLUSION

We describe SPHINX, a controller agnostic tool that leverages flow graphs to detect security threats on network topology and data plane forwarding originating within SDNs. We show that existing controllers are vulnerable to such attacks, and SPHINX can effectively detect them in realtime. SPHINX incrementally builds and updates flow graphs with succinct metadata for each network flow and uses both deterministic and probabilistic checks to identify deviant behavior. Our evaluation shows that SPHINX imposes minimal overheads.

ACKNOWLEDGEMENT

We thank our shepherd, Guofei Gu, and the anonymous reviewers for their valuable comments. We are also grateful to Anil Vishnoi, Dhruv Sharma, and Vinod Ganapathy for their feedback on an earlier draft of the paper.

REFERENCES

[1] "ARP poisoning attack," http://goo.gl/p4AVhf.
[2] "Cbench," http://www.openflowhub.org/display/floodlightcontroller/Cbench+(New).
[3] "CRATE datasets," ftp://download.iwlab.foi.se/dataset.
[4] "Data Set for IMC 2010 Data Center Measurement," http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
[5] "Dynamic ARP Inspection," http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/dynarp.html.
[6] "Fake topology attack," http://goo.gl/zRG8bz.
[7] "LBNL/ICSI Enterprise Tracing Project," http://www.icir.org/enterprise-tracing/.
[8] "Maestro," https://code.google.com/p/maestro-platform/.
[9] "Mausezahn," http://www.perihel.at/sec/mz/.
[10] "Mininet," http://mininet.org/.
[11] "Netty," http://netty.io/.
[12] "Open DOVE," https://wiki.opendaylight.org/view/Open_DOVE:Main.
[13] "Open vSwitch," http://openvswitch.org/.
[14] "OpenDaylight," http://www.opendaylight.org/.
[15] "OpenFlow switch specification," http://openflow.org/documents/openflow-spec-v1.1.0.pdf.
[16] "POX," http://www.noxrepo.org/pox/about-pox/.
[17] "Project Floodlight," http://www.projectfloodlight.org/floodlight/.
[18] "Tcpreplay," http://tcpreplay.synfin.net/.
[19] "VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-05.
[20] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures," in *SafeConfig'10*.
[21] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, "Network Configuration in A Box: Towards End-to-End Verification of Network Reachability and Security," in *ICNP'09*.
[22] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test Openflow Applications," in *NSDI'12*.
[23] S. Chowdhury, M. Bari, R. Ahmed, and R. Boutaba, "PayLess: A Low Cost Network Monitoring Framework for Software Defined Networks," in *IEEE NOMS'14*.
[24] N. Feamster and H. Balakrishnan, "Detecting BGP Configuration Faults with Static Analysis," in *NSDI'05*.
[25] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *ICFP'11*.
[26] A. Guha, M. Reitblatt, and N. Foster, "Machine-Verified Network Controllers," in *PLDI'13*.
[27] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *NDSS'15*.
[28] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis," in *NSDI'13*.
[29] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *NSDI'12*.
[30] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-wide Invariants in Real Time," in *NSDI'13*.
[31] R. Kloti, "OpenFlow: A Security Analysis," Master's thesis, ETH, Zurich, 2012.
[32] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards Secure and Dependable Software-Defined Networks," in *HotSDN'13*.
[33] LBNL, "arpwatch," http://ee.lbl.gov/.
[34] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the Data Plane with Anteater," in *SIGCOMM'11*.
[35] 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., April 2008*.
[36] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A Compiler and Run-time System for Network Programming Languages," in *POPL'12*.
[37] S. Natarajan, X. Huang, and T. Wolf, "Efficient Conflict Detection in Flow-Based Virtualized Networks," *ICNC'12*.
[38] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A Security Enforcement Kernel for OpenFlow Networks," in *HotSDN'12*.
[39] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale Monitoring and Control for Commodity Networks," in *SIGCOMM'14*.
[40] G. P. Reyes, "Security assessment on a VXLAN-based network," Master's thesis, University of Amsterdam, Amsterdam, 2014.
[41] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *NDSS'13*.
[42] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks," in *CCS'13*.
[43] A. Voellmy and P. Hudak, "Nettle: Taking the Sting out of Programming Network Routers," in *PADL'11*.
[44] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On Static Reachability Analysis of IP Networks," in *INFOCOM'05*.