Automated Discovery of Cross-Plane Event-Based Vulnerabilities in Software-Defined Networking

Benjamin E. Ujcich*, Samuel Jero†, Richard Skowyra†, Steven R. Gomez†,
Adam Bates*, William H. Sanders*, and Hamed Okhravi†
*University of Illinois at Urbana-Champaign, Urbana, IL, USA
†MIT Lincoln Laboratory, Lexington, MA, USA

***Abstract*—Software-defined networking (SDN) achieves a programmable control plane through the use of logically centralized, event-driven controllers and through network applications (apps) that extend the controllers' functionality. As control plane decisions are often based on the data plane, it is possible for carefully crafted malicious data plane inputs to direct the control plane towards unwanted states that bypass network security restrictions (*i.e., cross-plane attacks*). Unfortunately, because of the complex interplay among controllers, apps, and data plane inputs, at present it is difficult to systematically identify and analyze these cross-plane vulnerabilities.**

We present EVENTSCOPE, a vulnerability detection tool that automatically analyzes SDN control plane event usage, discovers candidate vulnerabilities based on missing event-handling routines, and validates vulnerabilities based on data plane effects. To accurately detect missing event handlers without ground truth or developer aid, we cluster apps according to similar event usage and mark inconsistencies as candidates. We create an event flow graph to observe a global view of events and control flows within the control plane and use it to validate vulnerabilities that affect the data plane. We applied EVENTSCOPE to the ONOS SDN controller and uncovered 14 new vulnerabilities.

I. INTRODUCTION

Software-defined networking (SDN) has experienced a rapid rise in adoption within data center providers, telecommunication providers, and other enterprises because of its programmable and extensible control plane [30]. SDN claims to decouple the network's decision-making about forwarding (*i.e.,* the *control plane*) from the traffic being forwarded (*i.e.,* the *data plane*) so as to allow centralized oversight through an *SDN controller* and *network applications* (or *apps*) in the enforcement of consistent (security) policies.

All popular modern SDN controllers, including ONOS [7], OpenDaylight [51], Hewlett Packard Enterprise's VAN SDN Controller [21], and Floodlight [17], operate as reactive *event-driven architectures* that, based on data plane activities, use asynchronous event dispatchers, event listeners, and controller API calls to pass information among controller and app components.[1] Each app's event listeners subscribe to a subset of the possible universe of events. Based on the event, an app may call API services (*e.g.,* a request to insert a new flow rule) or generate new events (*e.g.,* a notification that a new host has been seen in the data plane).

SDN's programmability significantly alters the control plane's attack surface. The claim of control and data plane decoupling belies a subtle and serious challenge: control plane decisions are often made as a result of information collected from an untrustworthy data plane. Prior attacks [14], [22], [57] have demonstrated specific examples of what we generalize as the class of *cross-plane attacks*, which allow attackers to influence control plane decision-making without attacking the controller or apps directly [70]. For instance, a clever attacker who controls a data plane host can emit packets that are acted upon by controller and app components, which can result in malicious privilege escalation or malicious control over flow rule behaviors by a host.

In the context of cross-plane attacks, decisions made based on untrusted data plane input may cause event handlers to execute unintended code paths, or prevent the execution of intended code paths, within the controller or apps. The event-driven, composable, and interdependent nature of controller and app components provides new potential for vulnerabilities based on which apps handle (or, critically, which apps *do not* handle) different kinds of events. For instance, apps that operate as intended in isolation may create conflicting behaviors when used together, and that may create vulnerable conditions that are not found when apps are used in isolation. As a result, the security posture of the SDN control plane does not rely on properties of individual controller or app components, but rather on the system-wide behavior of the components' event interactions as a whole.

The vulnerabilities that result from complex event and app interactions are challenging to detect automatically because such vulnerabilities are a class of *logic* (or *semantic*) *bugs* that require local and global semantic understanding about events and their use. Logic bugs are of interest to attackers because such bugs are difficult to identify during software development and can persist for years before disclosure [40]; existing tools often focus on bugs related to language grammar or resource

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24080
www.ndss-symposium.org

[1] An SDN controller service or app often consists of multiple functional units, which we call *components*. A functional unit ends at an API boundary or event dispatch.

use only (*e.g.,* FindBugs [23], PMD [53], and Coverity [8]) or require developers to annotate code (*e.g.,* KINT [63]), rendering such tools difficult to use in practice [27], [58].

In the absence of developer annotations that specify intended app behavior, the vulnerability search space can become large [11], [55], [34] [27]. However, by focusing on a narrower scope of event-related vulnerabilities that involve *missing* or *unhandled events*, we can tractably enumerate those conditions and investigate them. Uncovering such vulnerabilities requires understanding of how events are used within SDN components, how events are passed between SDN components, and how events' actions propagate within the control plane to have data plane effects. Given the event-driven nature of modern SDN architectures, our insight is that *event-related bugs that result from unhandled events are of high interest in SDN security evaluation*, particularly if cross-plane attacks can be used to trigger such vulnerabilities that ultimately lead to data plane consequences (e.g., flow rule installation).

Although tools have been developed to perform vulnerability discovery in SDNs with fuzz testing [25], [34], concurrency detection [66], and code analysis [32], [33], we are not aware of any tools that are designed specifically to aid developers and practitioners in the understanding of global event use and in the identification of unhandled event vulnerabilities at design and testing time. Forensic SDN tools [62], [59] provide causal explanations of past executions but do not identify vulnerabilities ahead of time.

Overview: In this paper, we propose a systematic approach for discovering cross-plane event-based vulnerabilities in SDN. We designed a tool, EVENTSCOPE, that aids practitioners and developers in identifying candidate vulnerabilities and determining whether such vulnerabilities can manifest themselves in the context of apps currently in use. Rather than discover the existence of "bad" events, our goal is to identify where the absence of a certain event handler may prevent developer-intended code paths from executing. We investigate how SDN controllers and apps use events to influence *control flow* (*i.e.,* the series of code paths in the control plane that are or are not executed) as well as implicit *data flow* (*i.e.,* the propagation of untrusted data plane input that may impact control plane decisions).

Our initial challenge is to identify what events an app should handle. It is complicated because no ground truth exists for this task, making simple heuristics and supervised learning techniques difficult to apply. A naïve solution would be to require an app to handle all events, but there are instances in which an app does not need to do so, *i.e.,* the lack of handling of certain events does not negatively impact the app's expected operation or cause deleterious data plane effects. Instead, EVENTSCOPE analyzes how events are handled within apps' event listeners relative to other apps to identify potentially missing events.

EVENTSCOPE then uses static analysis to abstract the SDN's API functionality and event flow into what we call an *event flow graph*. This data structure shows the control and data flow beginning from data plane inputs and ending at data plane outputs (*e.g.,* flow rule installation and removal). That allows EVENTSCOPE to identify the impact of a given component on other components in the system.

Using the event flow graph, EVENTSCOPE then validates whether potentially missing events can cause data plane effects in the presence or absence of other apps. Given an app with such a candidate vulnerability, EVENTSCOPE identifies other apps that handle that app's missing event and also have data plane effects to create a *context* for that vulnerability. Next, EVENTSCOPE represents these code executions as event flow graph paths to determine whether they have data plane effects. Finally, EVENTSCOPE generates a list of vulnerabilities for analysis by developers and practitioners.

Throughout this paper, we use the open-source, Java-based ONOS SDN controller [7] as a representative case study. ONOS is used in production settings by telecommunications providers, and its codebase underlies proprietary SDN controllers developed by Ciena, Samsung, and Huawei [45]. ONOS's extensive event-centered design makes the controller an ideal candidate for study. We analyzed how ONOS's core service and app components use events, discovering that many events are not handled even when components subscribe to those events. Although we focus on ONOS as a case study, we note that all modern SDN controllers use a similar event-based architecture; thus, EVENTSCOPE's methodology is broadly applicable to all such controllers.

We identify 14 new vulnerabilities in ONOS and, for selected cases, we show, through crafted exploits, how attackers are able to influence control plane behavior *from the data plane alone*. For instance, we were able to prevent ONOS's access control (firewall) app from installing flow rules, which allows hosts to communicate with each other in spite of access control policies that should have denied their communication (CVE-2018-12691). Additionally, we were able to leverage ONOS's host mobility app to remove the access control app's existing flow rules (CVE-2019-11189). These results demonstrate that, in real SDN implementations, instead of apps acting constructively and composably they often have competing and conflicting behavior. That conflict provides subtle opportunities for vulnerabilities to appear.

Contributions: Our main contributions are:

1) An automated approach to analyze **event use** by applications that identifies likely missing event handling and checks whether this lack of event handling can cause data-plane effects in combination with other apps.
2) The **event flow graph** data structure, which allows for succinct identification of (a) event dispatching, event listening, and API use among SDN components, as well as (b) the context to realize vulnerabilities.
3) An **implementation** of our vulnerability discovery tool, EVENTSCOPE, in Java and Python.
4) The discovery and validation of **14 new vulnerabilities** in ONOS that escalate data plane access.

II. BACKGROUND

We describe here the features of the SDN architecture. Although we use the ONOS SDN controller as a running example, we note that other SDN controllers (*e.g.,* Floodlight [17]) share similar event-driven features. We outline the challenges and mitigation approaches for SDN security that are related to

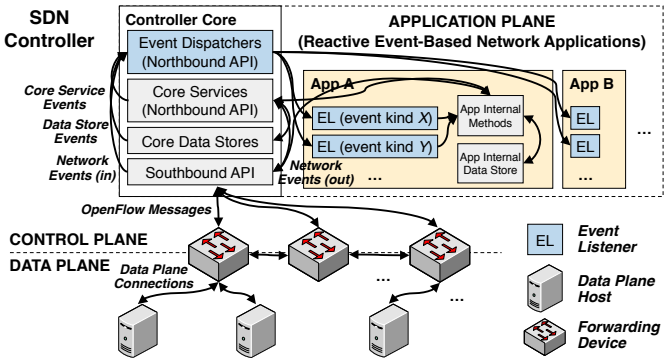


Fig. 1: SDN architecture overview. Apps subscribe to event dispatchers and implement event listeners. Network, data store, and service updates generate events.

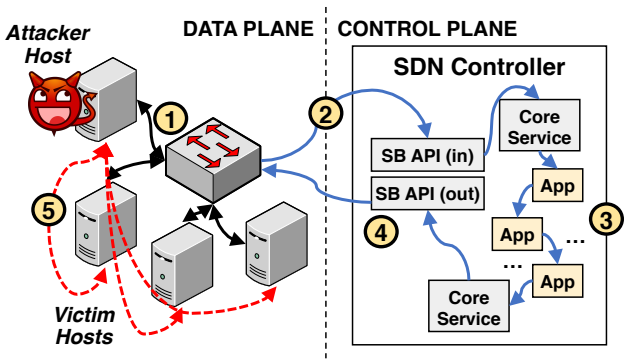


Fig. 2: Cross-plane attack example. Black arrows denote data plane connections, blue arrows denote control plane control flow, and red arrows denote intended effect (*e.g.,* increased data plane access). 1: An attacker emits data plane packets. 2: The controller's southbound API receives packets. 3: The controller's components use the data plane input to make a data plane decision. 4: The controller emits new packets or flow rules into the data plane. 5: The attacker uses the new packets or flow rules as a step to actuate an attack.

adversarial data plane input, event-driven apps, and event flow interactions.

A. SDN Architecture

1) Overview: Figure 1 shows an overview of the SDN architecture. SDN decouples how traffic decisions are made (*i.e.,* the *control plane*) from the traffic being forwarded (*i.e.,* the *data plane*). Traffic decisions are made in a logically centralized *controller* that functions as the core of a network operating system. Controllers manage network configurations and forwarding rules in the network's forwarding devices through the *southbound API* (*e.g.,* OpenFlow [39]).

2) Core services and app model: Controllers provide *core services* (*e.g.,* a host service that maintains data plane host information) as a basis for extended functionality through *network applications* (*apps*). Apps interact with the controller's core services through the *northbound API* and can be installed as reactive components within the controller or operate independently as proactive components that use RESTful interfaces. (See Appendix A for an example of the ONOS app's code.) Core services and apps use event listeners (described further in Section II-A3) to respond to events and to actuate further functionality by calling core or internal methods. The SDN app ecosystem allows third party and independent developers to write apps that can be installed in SDN controllers, and that can introduce security issues if apps are malicious [33], [32] or if apps serve as indirect conduits by which malicious activities can occur [59].

In addition to controlling forwarding decisions, SDN controllers also expose abstractions of network objects and processes. For instance, ONOS includes abstractions for Host objects that represent end hosts, and for Device objects that represent forwarding devices. Those abstractions are built on top of information learned or programmed from lower levels. In ONOS, the host location provider builds Host objects based upon information learned from Packet objects' header information. Apps interested in changes to hosts can reason about such changes at the level of a host abstraction rather than a packet abstraction.

3) Event model: Most SDN implementations are event-driven systems that model data plane changes as asynchronous *events*, such as the processing of an incoming data plane packet, the discovery of new network topology links, and changes in forwarding device states. Events have different *kinds* depending on the abstraction they describe (*e.g.,* hosts, packets, links), and each event kind may have different *event types* that further describe the functional nature of the event (*e.g.,* host added, host removed).

Events are sent from *event dispatchers* and received through *event listeners*. For instance, the controller may dispatch a network link event to all apps that are interested in network link state changes (*i.e.,* all apps that register link event listeners). An app that cares about new network links can use that event to make decisions about what functionality to perform (*e.g.,* recalculation of bandwidth for QoS guarantees). Such an app can also gather information about what the control plane's state looks like in the present (*i.e.,* API read calls), request changes to the control plane (*i.e.,* API write calls), or notify other apps and core services asynchronously (*i.e.,* event dispatching). That process is repeated by other apps and core services that register event listeners and react to events, and the combination of such interactions forms the basis (and complexity) of the event-driven SDN architecture.

In ONOS, apps can access the control plane's state through API read calls (*e.g.,* `getHosts()`) or by registering to receive asynchronous events (*e.g.,* listening for HostEvent events). API write calls can trigger event dispatches. ONOS uses a special listener for data plane packet events, the PacketProcessor, that allows components to receive or generate data plane packets.

B. SDN Security Challenges

1) Malicious data plane input: By design, the SDN architecture decouples the control and data planes. However, control plane decisions are often made as a result of information gathered from data plane input, allowing attackers to influence

control plane behavior even if the controller and app infrastructures are assumed to be hardened. *Cross-plane attacks*, such as topology poisoning [22], [14], [57], impact control plane operations by causing denial-of-service or connectivity-based attacks. Figure 2 shows a representative example of a cross-plane attack that uses malicious data plane input to produce an unintended data plane effect.

Attackers can infer whether the network is non-SDN or SDN and which controller is being used in an SDN setting [4], [71]. Defenses to date, such as control plane causality tracking [59], [62], trusted data plane identities [26], and timing-based link fabrication prevention [57], are useful in preventing specific classes of attacks but are not designed for vulnerability discovery because they track specific execution traces as they occur rather than all possible execution traces prior to runtime. Current SDN vulnerability tools, such as BEADS [25] and DELTA [34], rely on fuzzing techniques that do not easily capture complex event-based vulnerabilities.

Although controllers that are written in safely typed languages (*e.g.,* Java) can mitigate unchecked data plane input, type safety does not completely prevent misuse. An attacker can try to leverage syntactically valid data that may be semantically invalid depending on its use. For instance, the IPv4 address `255.255.255.255` is syntactically valid, but there may be unintended consequences if a controller or app component attempts to use it as a host address.

Mitigation approach: EVENTSCOPE analyzes how malicious data plane input and cross-plane attacks can have cascading effects throughout controller components and apps as a result of unhandled event types (Section V). We demonstrate how that analysis allows us to identify ONOS app vulnerabilities (Section VII).

2) Event-driven apps: SDN controller services and apps can subscribe to events of interest with event listeners. However, not all event types of a particular event kind may be handled. In the absence of well-defined formal properties (*e.g.,* safety and liveness) that specify what an app's behavior ought to be, it is not easy to automatically determine what constitutes "correct" or "incorrect" behavior. As a result, it is difficult to find bugs that are syntactically correct but semantically incorrect regarding the intended app behavior, and difficult to determine how that behavior affects the data plane.

Network verification approaches [29], [11] require formal property specifications or do not scale beyond trivial controllers. CONGUARD [66] and DELTA [34] offer models for reasoning about the ordering of OpenFlow events, but such events are only one part in a complex, event-driven, network operating system that must consider additional (and often more sophisticated) network abstractions.

Mitigation approach: EVENTSCOPE uses a clustering approach to infer the intended application behavior based on the insight that apps that perform similar functionality are interested in similar kinds of events and event types (Section IV). EVENTSCOPE identifies cases in which a given app's event types are absent with respect to similar apps and evaluates whether these absences create vulnerabilities (Section V-B).

3) Event flow interactions: As apps can originate from different parties [59], assessment of system-wide "correct" behavior is complex when components closely collaborate and form event-driven dependencies. The event-driven SDN architecture allows flexible and composable development, with events helping to provide convenient abstractions and allowing components to subscribe to asynchronous activities of interest. Prior work [1], [18], [43], [61], [42] has approached controller design by providing formally specified runtime languages and safe-by-construction controllers, but such approaches do not offer the extensibility of the operating-system-like controllers used in practice in production.

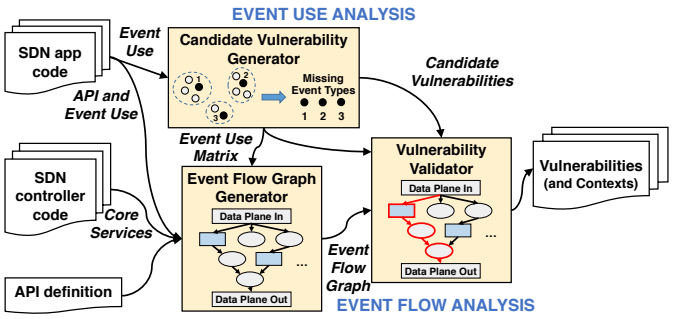


Fig. 3: EVENTSCOPE architecture overview.

Understanding how event-driven components in an SDN interact is challenging because events have both control flow and data flow elements. Events represent *control flow* because they are processed by event listener methods that may call additional methods depending on the event information, and they represent *data flow* because they carry data describing the event (*e.g.,* a host event contains that host's details). Although control flow and data flow can be modeled together in program dependence graphs [16] or code property graphs [67], analysis is often limited to single procedures because too many details prevent the analysis from scaling to complex, inter-procedural event-driven systems. Further, events can be used to influence what code paths are or are not taken and to trigger additional events.

Mitigation approach: EVENTSCOPE uses the event flow graph to model the key features of an event-driven SDN system while abstracting away unnecessary control flow details (Section V-A). The event flow graph shows how triggered events have consequences elsewhere, particularly when malicious data plane inputs later influence data plane changes.

III. EVENTSCOPE OVERVIEW

We designed EVENTSCOPE to identify cross-plane event-based vulnerabilities in three phases, as illustrated in Figure 3.

The first phase, the *candidate vulnerability generator*, takes the set of SDN apps as input and produces a list of unhandled event types for each app. In our implementation, we require the apps' Java bytecode. As ground truth about which event types apps should handle is not available, EVENTSCOPE uses a clustering approach that reports event types that are common in the cluster but are not handled in a particular app.

The second phase, the *event flow graph generator*, takes the apps' code, the controller's code, and a definition of controller API calls as inputs and constructs an event flow graph that

records how events propagate and influence the system. This includes event propagation within the controller as well as within apps and combinations of apps.

Finally, the event flow graph and the unhandled event types from the first two phases are combined in the third phase, the *vulnerability validator*, to identify the data plane impacts of unhandled event types. The output of this phase results in a list of vulnerabilities that can influence the data plane as a result of unhandled event types.

EVENTSCOPE automates the process and the phases work together, but for illustrative purposes, we discuss each of EVENTSCOPE's three phases separately before discussing the results from applying EVENTSCOPE to the ONOS SDN controller. In summary:

- The **candidate vulnerability generator (Section IV)** generates a list of possible vulnerabilities resulting from unhandled events based on apps' event use in comparison to that of similar apps.
- The **event flow graph generator (Section V-A)** analyzes the use of events between components to construct a concise representation of how events are passed and how they affect data plane operations.
- The **vulnerability validator (Section V-B)** filters and validates the possible missing-event-handling vulnerabilities from the first component by using the event flow graph to determine whether the missing event has had data plane impacts, either in isolation or in combination with other apps.

Inputs: Users provide EVENTSCOPE with the controller's code and apps' code to be analyzed. In our implementation, this code is provided as Java bytecode. EVENTSCOPE also requires a definition of the controller's northbound (*i.e.,* application) interface, which is simply the set of method signatures that comprise the northbound API.

Outputs: EVENTSCOPE produces a list of vulnerabilities related to missing-event handling that can impact the data-plane and the contexts in which the vulnerabilities occur. Practitioners can investigate such vulnerabilities to report bugs or to determine if exploits can be realized.

IV. EVENT USE ANALYSIS

In this section, we analyze the use of event kinds and event types in SDN app components and focus on unhandled events as signs of potential vulnerabilities. From that information, EVENTSCOPE generates a list of *candidate vulnerabilities*.

A. Event Use Methodology

Given the lack of ground truth about how apps should handle event types, we approach the problem of identifying possible unhandled event types by analyzing the similarity of different apps' uses of events. EVENTSCOPE clusters similar apps together, and, for each app, marks the unhandled event types in that app (with respect to that cluster) as a candidate vulnerability.

Algorithm 1 Candidate Vulnerability Generation

Input: Apps A, event kinds E_K, event types E_T, threshold τ
Output: List of candidate vulnerabilities V_C, event use matrix M
Initialize: $M[i][j] \leftarrow$ `false`$; \forall i \in A, \forall j \in E_T$ ▷ Event use matrix $M_{A \times E_T}$
$D[i][j] \leftarrow 0; \forall i \in A, \forall j \in A$ ▷ Distance matrix $D_{A \times A}$
$\mathcal{V} \leftarrow A \cup E_T, \mathcal{E} \leftarrow \emptyset, \mathcal{G}_S \leftarrow (\mathcal{V}, \mathcal{E})$ ▷ SimRank graph $\mathcal{G}_S$
$V_C \leftarrow \emptyset$ ▷ Candidate vulnerability list V_C

1: **for each** $a \in A$ **do**
2: $T \leftarrow$ getHandledEventTypes(a)
3: **for each** $t \in T$ **do**
4: $M[a][t] \leftarrow$ `true`
5: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(a,t),(t,a)\}$
6: $S \leftarrow$ SimRank$(\mathcal{G}_S, A)$ ▷ Similarity matrix $S_{A \times A}$
7: **for each** $i \in S$ **do**
8: **for each** $j \in S[i]$ **do**
9: $D[i][j] \leftarrow 1 - S[i][j]$ ▷ Distance = 1− Similarity
10: $C \leftarrow$ hierarchicalCluster(D, τ) ▷ Set of app clusters C
11: **for each** $c \in C$ **do**
12: $u \leftarrow \emptyset$ ▷ Union of event types within cluster c
13: **for each** $a \in C$ **do**
14: $u \leftarrow u \cup M[a]$
15: **for each** $a \in C$ **do**
16: $d \leftarrow u \setminus M[a]$ ▷ Set difference d of cluster and app
17: **for each** $t \in d$ **do**
18: $k \leftarrow$ getEventKind(t, E_K, E_T)
19: **if** k is handled by a **then**
20: V_C.append$((a,t))$

1) Algorithm: We describe EVENTSCOPE's approach, shown in Algorithm 1. We assume a set of apps that contain event listeners, A; a set of event kinds, E_K (*e.g.,* HostEvent in ONOS); a set of event types, E_T (*e.g.,* HOST_ADDED in ONOS) that relate to the functional nature of event kinds in E_K; and a threshold, τ, used to determine the number of app clusters. For intermediate data structures, we generate an event use matrix, M, that shows how apps use event types; a distance matrix, D, that represents the "distances" between apps in terms of how they are related; and a bipartite directed graph, $\mathcal{G}_S$, that represents the relations between apps and event types.

The algorithm determines the event types that each app uses (lines 1–5). It does so using static analysis through the generation of a control flow graph (CFG) of the relevant event listener method. If a given event type is handled (line 2), it is marked in the event use matrix, M, (line 4) and in the bipartite graph, $\mathcal{G}_S$ (line 5). The algorithm then computes the SimRank similarity metric across $\mathcal{G}_S$ and reduces it to vertices of interest, or $A \subset \mathcal{V}$, to produce the similarity matrix, S (line 5). It then takes the inverse of the similarity metric to compute the distance metric (lines 7–9), and uses it to compute app clusters by using a complete-linkage[2] (*i.e.,* maximum linkage) hierarchical clustering algorithm (line 10).

After the apps are partitioned into clusters, the algorithm inspects each app relative to its own cluster (lines 11–20). For each cluster, it generates a union of event types handled by that cluster's apps (lines 12–14). For a given app, it computes

[2]Alternatives include single-linkage and average-linkage clustering. We chose complete-linkage clustering because it 1) maximizes the distance between two elements of different clusters and 2) avoids the problem of grouping dissimilar elements that single-linkage clustering would entail [28].

what event types are not handled by that app's event listener with respect to the cluster's union (line 16). In some cases, the event type will be related to an event kind that the app does not handle at all, and we do not consider such scenarios to represent candidate vulnerabilities. When the event type's kind *is* handled by the app (line 19), the algorithm marks the event kind as a candidate vulnerability (line 20).

2) Design decisions: Initially, we applied the Levenshtein distance as our distance metric by treating each row of M as a bit vector, based on prior work on SDN app API use similarity [32]. However, we found that the Levenshtein distance did not capture the structural similarities among apps, event kinds, and event types. Instead, we opted for the SimRank metric, which expresses the idea that "two objects are similar if they are related to similar objects" [24]. SimRank fits more naturally with our problem of expressing the similarity of two apps that have relations to similar event types.

As each app includes a self-defined category, we were interested in whether such categories could describe functional event use similarity. However, we found that the categories are too vague to be meaningful for similar-event-handling identification, so we opted instead for a distance-based clustering approach that can be generated even if app categories are not specified. One example of the problem is that of the forwarding app fwd and the routing app routing in ONOS, which are both in the traffic engineering category. While we might expect those apps to be similar, since they are in the same category and share the same high-level objective of making traffic engineering decisions at different OSI layers, it turns out that the reactive forwarding app responds to new packets to make its decisions, while the routing app uses the existing network state to make its decisions. Those functional differences result in use of radically different event kinds and types.

3) Interpretation: Because apps do not provide well-defined semantics about their correct operation, we do not have ground truth about what event types each app *should* handle. As a result, we chose to focus on instances of missing event handling, which we can identify based on knowledge about the complete set of events. Unfortunately, such instances do not tell us the extent to which such missing events are intentional or the extent to which missing events' exploitation can cause unexpected behavior. While any instance is arguably a concern, we wanted to focus our effort on the instances *most likely* to be vulnerabilities. As a result, we chose to cluster apps in order to identify the missing event handling that stands out as the most "unusual," with the parameter τ approximating the unusualness of missing event handling.

As such, event use analysis can be viewed as a filtering step that attempts to identify the *most likely* unhandled event types for candidate vulnerabilities among all potential unhandled event types. EVENTSCOPE can be configured to be conservative and mark all unhandled event types as potential bugs; doing so requires setting $\tau = 1.0$ to generate 1 cluster.

B. Event Use Results

We evaluated EVENTSCOPE's event use analysis using ONOS v1.14.0 [46]. In addition to ONOS's core services, the ONOS codebase includes third-party apps written by indepen-

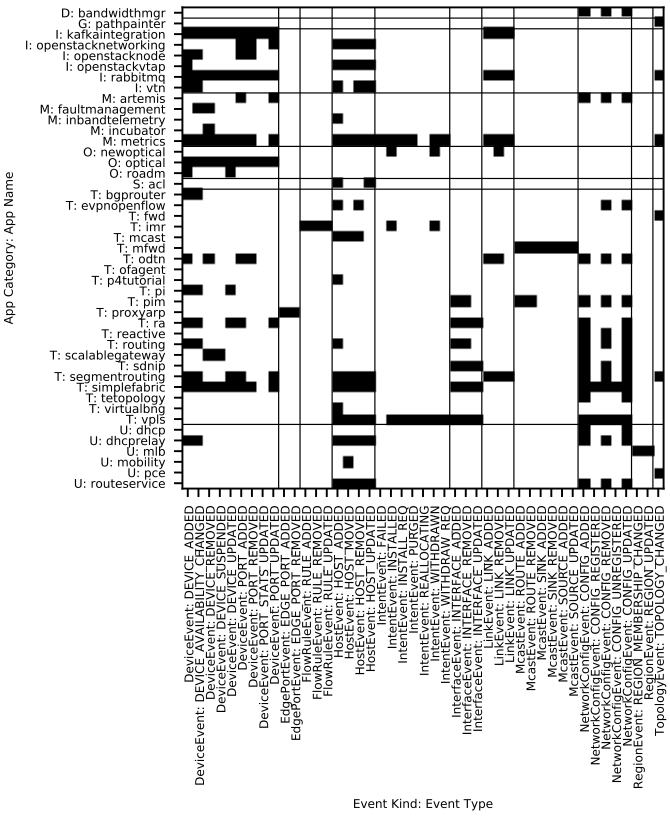


Fig. 4: ONOS event use matrix, M. Black cells represent event types that are handled by `event()` methods. Horizontal dividers represent app categories, and vertical dividers represent event kinds. (App category key: D = default, G = GUI, I = integration, M = monitoring, O = optical, S = security, T = traffic engineering, and U = utility.)

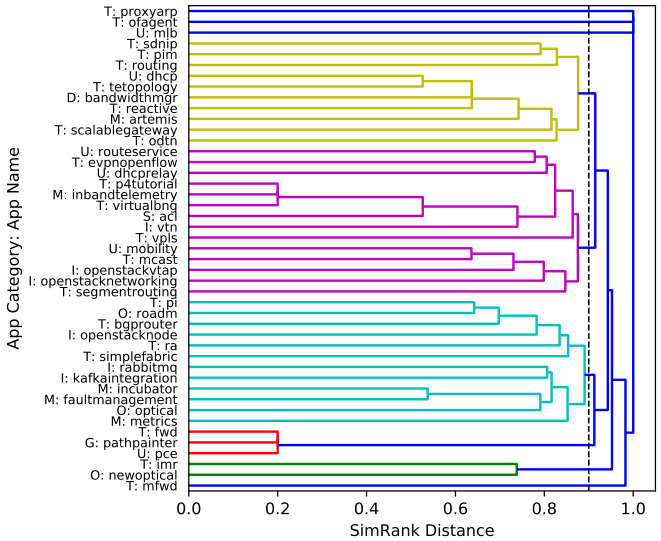


Fig. 5: Dendrogram representation of ONOS network event type similarity among apps, based on the SimRank distance metric. The dashed vertical line represents a threshold $\tau = 0.90$ with a partitioning of 9 clusters.

dent developers. We explain each part of the methodology as applicable to ONOS and its apps.

1) ONOS's event system: ONOS events implement the Event interface; they include `subject()` and `type()` methods that describe what the event is about (*e.g.,* a Host) and what type the event is, respectively. ONOS events are used for various subsystems, so we limit our study to network-related events only.[3]

We found that ONOS contains 95 network event listeners across 45 apps' event listeners.[4] Popular event kinds handled were DeviceEvent (25 instances), NetworkConfigEvent (22 instances), and HostEvent (18 instances). Overall, we found 45 event types among 11 (network) event kinds.

For each app's event listeners, we used static analysis on the listeners' bytecode to generate control flow graphs (CFGs) of any event handlers (*i.e.,* `event()` methods) within that app. Within each method, we considered an event type handled if it results in the call of other functional methods; we considered an event type to be not handled if it only executed non-functional methods (*e.g.,* logging) or immediately returned.

2) ONOS unhandled event types: Figure 4 shows EVENTSCOPE's generated event use matrix M of the 45 apps included with the ONOS codebase. Each ONOS app includes a self-defined category, and categories are grouped by horizontal dividers. Each event kind is grouped by vertical dividers. Figure 5 shows the dendrogram of the resulting app clusters, based on SimRank distance and complete-linkage clustering.

We empirically chose a threshold ($\tau = 0.90$) that yielded a number of clusters (*i.e.,* 9) similar to the number of categories of ONOS apps (*i.e.,* 8) based on the assumption that there exist at least as many categories as there are functional differences among apps. We found that that threshold worked well in the rest of our evaluation. (See Appendix C for an evaluation of τ on detection rates.) We found that setting the threshold too low (*i.e.,* more clusters) created more singleton app clusters, which should be avoided because each cluster's union of event types becomes the event types the app handles. However, setting the threshold too high (*i.e.,* fewer clusters) clustered apps with too few functional similarities. Based on that threshold, we generated 116 candidate vulnerabilities, which were used as input into the next stage of EVENTSCOPE (Section V).

V. EVENT FLOW ANALYSIS

Given a list of candidate vulnerabilities, we identify which vulnerabilities are reachable from the data plane and affect the data plane. To do so, we generate an event flow graph that shows how apps and the controller use events, and how these usages of events can interact to generate control flow in the control plane. Using that graph, we then validate our candidate vulnerabilities by analyzing how they impact subsequent control plane and data plane operations, looking for impacts in the control plane that can be caused by other data plane events. That results in a list of vulnerabilities with real impacts on the data plane.

A. Event Flow Graph Generation

In order to determine reachable candidate vulnerabilities from the data plane that affect the data plane (via the control plane), EVENTSCOPE uses static analysis to create an *event flow graph* that illustrates how events and API calls propagate from the data plane to the controller and apps.

1) Definitions: We formalize a *component* as a fragment of the SDN codebase that begins at an event listener method or core service method and ends at an API boundary or event dispatch. An app or core service can have more than one component if it has more than one event listener. As a result of that definition, each component serves as an *entry point*[5] into control plane functionality. Our objective is to determine the fragments of controller and app code that are reachable from each entry point.[6]

Formally, an *event flow graph*, denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a directed, multi-edged graph that models the abstractions for inter-procedural and inter-component control and data flows in the SDN control plane. Event flow graphs summarize the necessary control and data flows among components needed for event flow analysis. Vertices, denoted by $\mathcal{V}$, consist of one of the following types: event listeners (represented as entry point methods), API services (represented as an API interface method or its implemented concrete method), and representations of data plane input (`DPIn`) and data plane output (`DPOut`). Edges, denoted by $\mathcal{E}$, are labeled and consist of one of the following types: API read calls (`API_READ`), API write calls (`API_WRITE`), data plane inputs to methods (`DP_IN`), methods' output to the data plane (`DP_OUT`), or passing of an event type (*e.g.,* HOST_ADDED event type of the HostEvent event kind).

EVENTSCOPE uses a two-phase process in which it first examines which events are used within each app and then considers how these events propagate and cause other events in the context of multiple apps. As a result, EVENTSCOPE's event flow graph can represent multiple apps as well as dependencies among apps. The dependencies among applications for event processing are shown as edges in the event flow graph. One event that is processed by multiple applications (*i.e.,* event listeners) is represented as a node with multiple outgoing labeled edges with the respective event type; each edge is directed towards an event listener of that event kind.

2) Methodology: EVENTSCOPE's approach is shown in Algorithm 2. It initializes the event flow graph's vertices to be the set of event listeners and representations for data plane inputs and outputs. It begins with the set of event listeners as the components of entry points to check (line 1). For each entry point, it generates a call graph (line 5). Within the call graph, it checks whether calls relate to an API read (lines 7–10), to an API write (lines 11–14), or to the event dispatcher to generate new events (lines 15–16). It links the event dispatchers and event listeners together in the event flow graph by using the event use matrix, M, generated in the prior step (Section IV); each event type that is handled by

[3] Event implementation classes with the prefix org.onosproject.net.*.

[4] We note that ONOS core service components also include event listeners for inter-service notifications. We did not evaluate those listeners' event uses because we assume that all event types handled by each core service event listener are the event types necessary for correct functionality.

[5] In traditional static analysis, a program has a well-defined entry point: the `main()` function. However, since SDN is event-driven, no `main()` function exists [62]. To correct for the lack of a `main()` function and to account for the event-driven architecture, we use each component as an entry point.

[6] Lu *et al.* [38] define that as "splitting" in the component hijacking problem.

Algorithm 2 Event Flow Graph Generation

Input: API read methods A_r, API write methods A_w, data plane input methods D_i, data plane output methods D_o, event listener methods E_l, event kinds E_K, event types E_T, event use matrix M
Output: Event flow graph $\mathcal{G}$
Initialize: $\mathcal{V} \leftarrow E_l \cup \{\texttt{DPIn}, \texttt{DPOut}\}$, $\mathcal{E} \leftarrow \emptyset$
$S \leftarrow E_l$ ▷ Stack S of entry points (*i.e.,* components) left to check
$C \leftarrow \emptyset$ ▷ Checked components C
$E_d \leftarrow \emptyset$ ▷ Components that dispatch events E_d
1: **while** S is not empty **do**
2: $e \leftarrow S$.pop ▷ Entry point method e
3: **if** $e \in C$ **then**
4: **continue** ▷ Skip entry point if already processed
5: $(c_v, c_e) \leftarrow$ generateCG(e) ▷ Call graph vertices c_v and edges c_e
6: **for each** $c \in c_v$ **do**
7: **if** $c \in A_r$ **then**
8: $\mathcal{V} \leftarrow \mathcal{V} \cup \{c\}$
9: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(c, e)\}$ ▷ Labeled edge `API_READ`
10: S.push(c)
11: **else if** $c \in A_w$ **then**
12: $\mathcal{V} \leftarrow \mathcal{V} \cup \{c\}$
13: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(e, c)\}$ ▷ Labeled edge `API_WRITE`
14: S.push(c)
15: **else if** c is the event dispatch method **then**
16: $E_d \leftarrow E_d \cup \{(c)\}$
17: $C \leftarrow C \cup e$
18: $\mathcal{E} \leftarrow$ linkListenersDispatchers$(\mathcal{E}, E_d, E_l, E_K, E_T, M)$ ▷ Labeled edges of particular event type $t \in E_T$
19: $\mathcal{E} \leftarrow$ linkDataPlane$(\mathcal{E}, D_i, D_o)$ ▷ Labeled edges `DP_IN` or `DP_OUT`
20: $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E})$

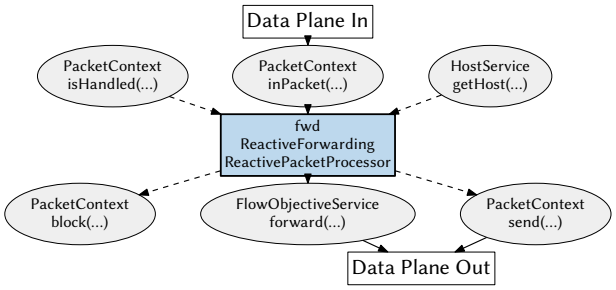


Fig. 6: Event flow graph of fwd's packet processor. Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, and dashed edges represent API calls.

a particular listener is represented as its own edge, so multi-edges are possible (line 18). Finally, it identifies core service components that take in data plane input or generate data plane output, and links those to the data plane input and output vertices (line 19).

3) Results: To show how an event flow graph abstracts useful information for understanding SDN architecture events, we consider the partial event flow graph from ONOS shown in Figure 6. It shows the forwarding app (fwd) packet processor component as an entry point. (For event flow graphs that include event dispatch edges, see Figures 8 and 9 in Section VII and Figure 11 in Appendix B.) General static analysis tools produce control flow graphs (CFGs) for each procedure or method, as well as a call graph (CG) for inter-procedural analysis; however, static analysis tools face challenges regarding the understanding of API behavior and the semantics of a given program's domain [58]. While both CFGs and a CG are necessary for control or data flow analyses, neither type of graph represents the SDN domain's semantics of events or API behavior at the right level of abstraction.

We generated an ONOS event flow graph whose components include core services, providers[7], and 45 apps. The ONOS event flow graph's nodes consists of representations of 143 event listeners, 25 packet processors, 81 API call methods of core services, 1 data plane input node, and 1 data plane output node. The ONOS event flow graph's edges consist of representations of 396 API calls, 352 event dispatches, and 21 data plane interactions. Appendix B shows a partial representation of that event flow graph based on 5 sample apps and the core services that they use.

Because ONOS does not specify a precise set of API calls that comprise the northbound API [59], we used the public method signatures of the *Service and *Provider classes, along with those methods' return values, to determine API read and write calls, resulting in 123 API read call methods, 87 API write call methods, 1 method directly related to data plane input, and 44 methods directly related to data plane output and effects. We identified event dispatching based on direct calls to the event dispatcher for local events (*e.g.,* `post()`) or indirect calls to a store delegate[8] for distributed events.

B. Vulnerability Validation

Now that we have an event flow graph, we can combine it with our candidate vulnerabilities to understand the extent to which unhandled event types have data plane consequences.

We focus on *valid vulnerabilities* as those in which the following conditions are met: 1) an app's event listener does not handle a particular event type, 2) that event listener can be called as a result of actions triggered from data plane input, and 3) in handling the other event types, that event listener can take some subsequent action that affects the data plane (*i.e.,* data plane output). In essence, we investigate the cases in which such an event handler would otherwise be affected by data plane input and have an effect on the data plane. Vulnerabilities defined in this way can be expressed as path connectivity queries in the event flow graph.

1) Context: Event handling vulnerabilities do not occur in isolation, but as part of a complex interaction web involving many other event handlers and apps We need to consider that context when discussing a given vulnerability. We borrow from Livshits and Lam [37] the intuition that exploitable vulnerabilities can occur as a result of a multi-stage exploit via an initial data injection and a subsequent app manipulation.

[7]In ONOS, a *provider* interacts with core services and network protocol implementations [48]. We consider provider services to be core services.

[8]ONOS uses distributed data stores across ONOS instances to store network state information. An instance can notify other instances of a change to the data store (*e.g.,* a MapEvent event update of a Host object modification in the host data store). That notification causes each instance to re-dispatch events locally (*e.g.,* a HostEvent event).

Algorithm 3 Vulnerability Validation

Input: Event flow graph $\mathcal{G}$, list of candidate vulnerabilities V_C, event use matrix M, apps A
Output: List of vulnerabilities and contexts V
Initialize: $V \leftarrow \emptyset$ ▷ Vulnerabilities and contexts list V
1: **for each** $(a, t) \in V_C$ **do** ▷ App $a \in A$ and event type $t \in E_T$
2: $E_l \leftarrow$ getEventListeners$(a, \mathcal{G})$ ▷ $E_l \subset \mathcal{G}$'s vertices
3: **if** $\neg$(pathExists(DPIn $\rightarrow e \in E_l \rightarrow$ DPOut, $\mathcal{G}$)) **then**
4: **continue**
5: $c_+ \leftarrow \emptyset, c_- \leftarrow \emptyset$ ▷ Present context set c_+, absent context set c_-
6: **for each** $a_i \in A \setminus \{a\}$ **do** ▷ All apps except a
7: $E_{l_i} \leftarrow$ getEventListeners$(a_i, \mathcal{G})$ ▷ $E_{l_i} \subset \mathcal{G}$'s vertices
8: **if** pathExists(DPIn $\rightarrow e \in E_{l_i} \rightarrow$ DPOut, $\mathcal{G}$) **then**
9: **if** $t \in$ getHandledEventTypes(E_{l_i}, M) **then**
10: $c_+ \leftarrow c_+ \cup a_i$
11: **else**
12: $c_- \leftarrow c_- \cup a_i$
13: V.append$((a, t, c_+, c_-))$

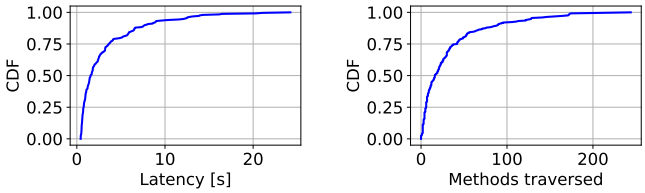
As a result, we define the *present context* as the set of other apps that 1) handle the vulnerability's missing event type in the absence of the vulnerable app's event handler's handling of it, 2) are affected by data plane input, and 3) have data plane effects. We define *absent context* as the set of other apps that, like the app in question, do not handle the vulnerability's missing event type but can be affected by data plane input and have data plane effects.

The present context lets us determine what the data plane effects are if the unhandled event type is dispatched. The absent context lets us determine what other apps might have concurrent influence over data plane effects. We note that context is necessary but not sufficient for *exploit generation*. Context is an over-approximation of the set of apps needed to exploit the vulnerability.

We note that exploit generation is nontrivial and that automatic exploit generation [3] is an ongoing research area. EVENTSCOPE's output includes "valid" vulnerabilities and contexts that EVENTSCOPE believes to be reachable from the data plane and to have data plane impacts. While EVENTSCOPE's validation provides strong soundness properties, static analysis is necessarily imprecise; manual verification is still recommended. EVENTSCOPE provides precisely the details that need to be included in a bug report. However, the tool neither provides a guarantee that a bug exists nor automatically submits bug reports. For the vulnerabilities EVENTSCOPE found, we manually examined the source code to confirm that the vulnerabilities existed.

2) Methodology: EVENTSCOPE's approach for vulnerability validation is shown in Algorithm 3. It uses the event flow graph, candidate vulnerabilities, and the event use matrix as inputs. Each candidate vulnerability is represented as a tuple of the app and unhandled event type (line 1). For each event type, EVENTSCOPE gets the app's event listeners (line 2). It performs path connectivity queries over the event flow graph. If at least one path does not exist that starts from the data plane, goes through one of the app's event handlers, and ends in the data plane, then the algorithm does not consider a vulnerability to be relevant, either because the event listener is not affected by data plane input or because the resulting path does not have data plane effects (lines 3–4).

The algorithm initializes the present and absent context sets to be empty (line 5). It inspects all of the other apps in the event flow graph to build the context (line 6). If another app's event listener is affected by data plane input and has data plane effects (line 8), it checks whether the missing event type is handled by that app (line 9) or not (line 11), and builds the context sets accordingly (lines 10 and 12). It then appends the vulnerability to the vulnerability list (line 13).



(a) Latency CDF (b) Method traversal CDF

Fig. 7: Component analysis performance results.

C. Performance Results

We ran EVENTSCOPE using an Intel Core i5-4590 3.30 GHz CPU with 16 GB of memory on ONOS and its associated apps. Figure 7 shows the cumulative distribution functions (CDFs) of the component analysis latency (Figure 7a) and the number of methods traversed in the call graph generation (Figure 7b); the latency corresponds to the computations of lines 2–17 in Algorithm 2, and the methods traversed correspond to line 5 in Algorithm 2.

In total, we analyzed 249 components found within ONOS's 1.2 million lines of Java, which required full traversals across 8064 method invocations for call graph generation. We found that the median per-component analysis time was 1.55 s and the mean per-component analysis time was 3.14 s, or approximately 13 min in total. For call graph generation, we found that each component required a median traversal of 16 methods and a mean traversal of 32 methods. We also measured EVENTSCOPE's peak memory consumption by using `time` and found that EVENTSCOPE used 1.82 GB of memory.

VI. IMPLEMENTATION

We implemented EVENTSCOPE using a combination of Python and Java. In Python, we used Scikit-learn [52] to perform hierarchical clustering in the event use analysis. In Java, we used Soot [60] to generate the control flow graphs and call graphs used for event use analysis and for determining entry points. Soot creates an intermediate representation in Jimple. We also used JGraphT [44] to store in-memory representations of event flow graphs and to query path connectivity. Source code for our implementation's components is available at https://www.github.com/bujcich/EventScope.

For connectivity queries in lines 3 and 8 in Algorithm 3 (*i.e.,* `pathExists()`), we used Dijkstra's algorithm. The worst-case performance time for each `pathExists()` query can be optimized [19] to $O(2\,(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|))$, where $|\mathcal{E}|$ represents the number of event flow graph edges and $|\mathcal{V}|$

represents the number of event flow graph nodes. In practice, we found that the small number of apps and events did not pose a challenge for connectivity computations.

Soot operates on Java bytecode, which allows EVENTSCOPE to analyze closed-source Java-based controllers and apps. Similar program analysis tools, such as `angr` [56], can operate on closed-source binary executables. Using bytecode is advantageous, as we can use EVENTSCOPE to generate event flow graphs without requiring Java source code. Thus, EVENTSCOPE can be useful for practitioners as a code audit tool. Although we did not encounter any apps that used dynamic calls, such as the Java language's reflection API, TamiFlex [10] extends Soot to perform static analysis that accounts for reflection.

Although our implementation generates a list of vulnerabilities for ONOS, EVENTSCOPE is not specific to ONOS. EVENTSCOPE's analysis and methodology can be applied to any event-driven SDN controller, which includes popular controllers such as OpenDaylight, HPE VAN, and Floodlight.

VII. ONOS VULNERABILITY EVALUATION RESULTS

EVENTSCOPE identified 14 vulnerabilities that satisfy all of the following properties: 1) the vulnerable event handler features an unhandled event type, which was identified through similarity clustering analysis; 2) the event handler can be reached from data plane input; and 3) the event handler can reach a data plane output.

Table I shows the 14 vulnerabilities, based on app, event kind, and unhandled event types. Table I also provides sample paths in the event flow graph. We found that all vulnerabilities involved the HostEvent event kind, which indicates that data plane input has the most effect on host information in ONOS.

EVENTSCOPE's output included 14 possible vulnerabilities. We manually investigated each vulnerability in the source code and determined that all of them could be exploited from the data plane. As a result, Table I represents EVENTSCOPE's complete output with no false positives. EVENTSCOPE's final phase essentially filters out missing event handling that cannot be reached from the data plane or trigger impacts on the data plane; as a result, the output provides strong soundness properties. As we do not have ground truth about which unhandled event types should be handled, we note that the event use analysis in Section IV-A should be interpreted as a filter of the unhandled event types that are *most likely* to require attention, based on such event types' absence *vis-à-vis* a cluster of the most similar apps. As noted earlier, we chose the clustering threshold that produced a number of clusters closely matched to the number of ONOS app categories.

We describe exploits for two of the vulnerabilities below in Sections VII-A and VII-B, and then, for the sake of space, briefly discuss the impact of the other vulnerabilities. For the exploits we created, we used a Mininet [31] SDN network. We wrote our exploit scripts in Python and used the Scapy [9] network packet library to generate data plane input.

We notified the ONOS Security Response Team of the vulnerabilities and exploits that we discovered through a responsible disclosure process. We explained the vulnerabilities and demonstrated working exploits.

A. Data Plane Access Control Bypass with acl and fwd (CVE-2018-12691)

1) Summary: We found that an attacker could bypass data plane access control policies by sending semantically invalid packets into the data plane to corrupt the controller's view of hosts. That prevented the access control app, acl, from installing flow deny rules, and that effectively bypassed the desired access control policy.

We assume a topology of at least two hosts: h1 and h2. The attacker controls host h1 and wants to communicate with h2. An access control policy prevents h1 and h2 from communicating.

2) Method: The attack occurs in two stages.

First, the attacker host h1 sends into the data plane an ICMP packet with an invalid source IP address (*e.g.,* the broadcast address). The host provider learns about host h1 from the ICMP packet's source MAC address, creates a host object (without an associated IP address), and generates a HostEvent event with a HOST_ADDED event type.[9] On the HOST_ADDED event type, acl checks whether flow deny rules should be installed for the added host. Since acl performs this check at the IP layer only and host h1 has an empty IP address list, no flow deny rules are installed.

Next, the attacker host h1 sends traffic intended for the target host h2. The host provider references the prior host object representing host h1, updates host h1's list of IP addresses with host h1's real IP address, and generates a HostEvent event with a HOST_UPDATED event type. Prior to patching the vulnerability, acl did not check for the HOST_UPDATED event type and took no action with such events. Another app, such as fwd, then installs flow allow rules from the attacker host h1 to the target host h2.

3) Results and implications: We wrote an exploit that performed the attack, and we were able to demonstrate that messages could be sent from the attacker to the target. From a defender's perspective, the exploit's effects may not be obvious immediately because the flow deny rules were never installed. A defender would need to check for evidence of the absence of the flow deny rules or the unintended presence of the flow allow rules. Since the host object corruption in the first stage need not occur at the same time as the lateral movement in the second stage, a stealthy attacker could wait until he or she needed to use such elevated access at a later time.

4) Event flow graph: Figure 8 shows the partial event flow graph with the relevant code paths used by the attacker. The attack's first stage follows the left-side path, in which the attack corrupts the host information in the HostProviderService. The attack's second stage triggers a HOST_UPDATED event type that does not get handled by acl's host event listener; in addition, the attack's second stage succeeds as shown by the right-side path.

[9]The HOST_ADDED event type assumes that the controller has never seen that host's MAC address before, but that is unlikely to be true if host h1 had sent any traffic prior to attacker compromise. However, if we assume that the attacker has root privileges on host h1, the attacker can change host h1's network interface MAC address. Thus, host h1 will appear as a newly added host and trigger the HOST_ADDED event type if the host subsequently sends any traffic into the data plane.

TABLE I: Event Listener Vulnerabilities Based on Event Flow Graph Analysis and Event Use Filtering ($\tau = 0.90$).

#	CVE ID	App	Unhandled type	Example event flow graph path showing potential data plane input to data plane effect
*	CVE-2018-12691	acl	HOST_UPDATED	See Figures 8 and 9 for event flow graph examples.
1	CVE-2019-11189	acl	HOST_MOVED	
2	CVE-2019-16300	acl	HOST_REMOVED	
3	CVE-2019-16298	virtualbng	HOST_MOVED	DPIn $\xrightarrow{\text{DP\_IN}}$ inPacket() $\xrightarrow{\text{API\_READ}}$ provider.host.InternalHostProvider $\xrightarrow{\text{API\_WRITE}}$ hostDetected() $\xrightarrow{\text{HOST\_ADDED}}$ virtualbng.InternalHostListener $\xrightarrow{\text{API\_WRITE}}$ startMonitoringIp() $\xrightarrow{\text{DP\_OUT}}$ DPOut
4	CVE-2019-16298	virtualbng	HOST_REMOVED	
5	CVE-2019-16298	virtualbng	HOST_UPDATED	
6	CVE-2019-16299	mobility	HOST_ADDED	DPIn $\xrightarrow{\text{DP\_IN}}$ inPacket() $\xrightarrow{\text{API\_READ}}$ provider.host.InternalHostProvider $\xrightarrow{\text{API\_WRITE}}$ hostDetected() $\xrightarrow{\text{HOST\_MOVED}}$ mobility.InternalHostListener $\xrightarrow{\text{API\_WRITE}}$ removeFlowRules() $\xrightarrow{\text{DP\_OUT}}$ DPOut
7	CVE-2019-16299	mobility	HOST_REMOVED	
8	CVE-2019-16299	mobility	HOST_UPDATED	
9	CVE-2019-16301	vtn	HOST_MOVED	DPIn $\xrightarrow{\text{DP\_IN}}$ inPacket() $\xrightarrow{\text{API\_READ}}$ provider.host.InternalHostProvider $\xrightarrow{\text{API\_WRITE}}$ hostDetected() $\xrightarrow{\text{HOST\_ADDED}}$ vtn.InternalHostListener $\xrightarrow{\text{API\_WRITE}}$ forward() $\xrightarrow{\text{DP\_OUT}}$ DPOut
10	CVE-2019-16302	evpnopenflow	HOST_MOVED	DPIn $\xrightarrow{\text{DP\_IN}}$ inPacket() $\xrightarrow{\text{API\_READ}}$ provider.host.InternalHostProvider $\xrightarrow{\text{API\_WRITE}}$ hostDetected() $\xrightarrow{\text{HOST\_ADDED}}$ evpnopenflow.InternalHostListener $\xrightarrow{\text{API\_WRITE}}$ forward() $\xrightarrow{\text{DP\_OUT}}$ DPOut
11	CVE-2019-16302	evpnopenflow	HOST_UPDATED	
12	CVE-2019-16297	p4tutorial	HOST_MOVED	DPIn $\xrightarrow{\text{DP\_IN}}$ inPacket() $\xrightarrow{\text{API\_READ}}$ provider.host.InternalHostProvider $\xrightarrow{\text{API\_WRITE}}$ hostDetected() $\xrightarrow{\text{HOST\_ADDED}}$ p4tutorial.InternalHostListener $\xrightarrow{\text{API\_WRITE}}$ applyFlowRules() $\xrightarrow{\text{DP\_OUT}}$ DPOut
13	CVE-2019-16297	p4tutorial	HOST_REMOVED	
14	CVE-2019-16297	p4tutorial	HOST_UPDATED	

* We note that we originally discovered CVE-2018-12691 manually, which led us to investigate event-based vulnerabilities and to create the EVENTSCOPE tool. We include CVE-2018-12691 here for completeness.

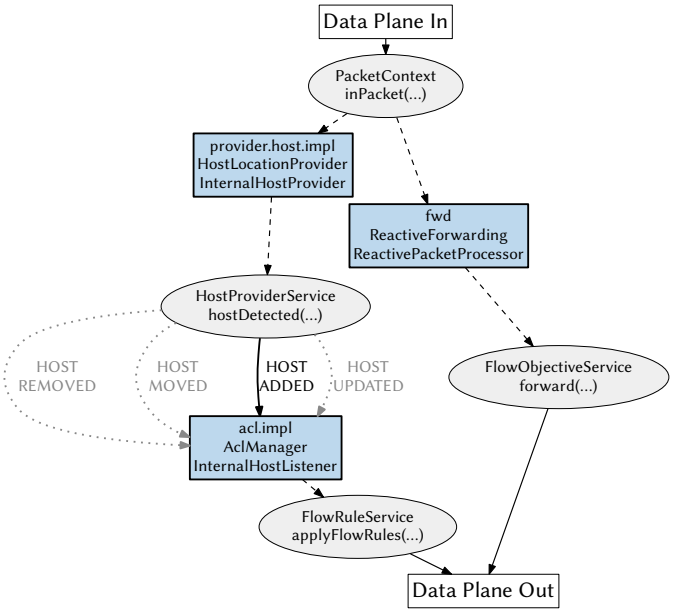


Fig. 8: Partial event flow graph showing vulnerable code paths used in CVE-2018-12691. Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, bold edges represent event dispatches, and dashed edges represent API calls. (Dotted gray edges represent unhandled event types, which are shown for reference.)

In the analysis of acl, EVENTSCOPE produces an absent context set, c_{-}, that includes fwd. The absent context set represents other event listeners and packet processors that might also respond to the same set of data plane input and produce data plane effects. A practitioner would discover that an app in the absent context set is producing undesirable effects via flow rule installation by fwd.

B. Data Plane Access Control Bypass with acl, mobility, and fwd (CVE-2019-11189)

1) Summary: We found that an attacker could bypass the data plane access control policies by spoofing another host using ARP reply packets. Such a spurious location change can allow the host mobility app, mobility, to remove acl's flow deny rules. Since acl does not reinstall such flow deny rules after a location change, the attacker can subvert network policy with increased access.

We assume a topology of at least three hosts: h1, h2, and h3. The attacker controls h1 and h3 and desires access to h2. Hosts h1 and h3 have different data plane connection points. An access control policy prevents communication between h1 and h2 as well as between h3 and h2.

2) Method: The attack occurs in two stages.

First, the attacker host h1 attempts to connect to the target host h2, but the connection is denied by acl's flow deny rules that were created when the hosts were detected or when a new access control policy was installed. The other attacker-controlled host, h3, sends into the data plane an ARP reply that spoofs the identity of host h1. The host provider determines

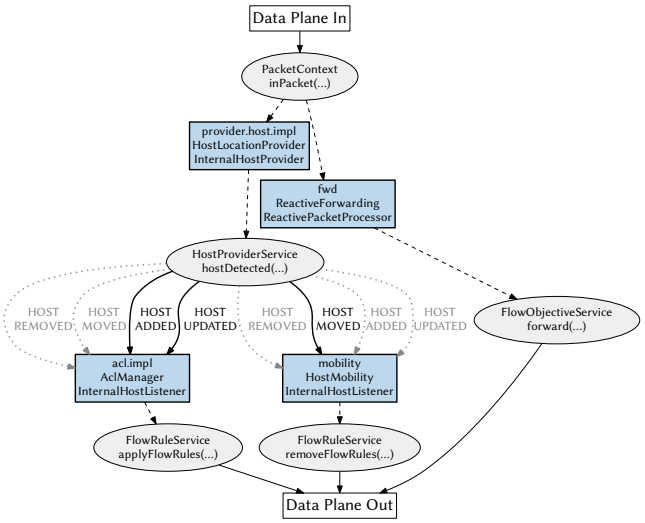


Fig. 9: Partial event flow graph showing vulnerable code paths used in CVE-2019-11189. Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, bold edges represent event dispatches, and dashed edges represent API calls. (Dotted gray edges represent unhandled event types, which are shown for reference.)

that host h1 has "moved" to the same connection point as host h3 and generates a HOST_MOVED event type. On the HOST_MOVED event type, mobility performs a network-wide cleanup that removes "old" flow rules whose source or destination MAC addresses match the respective host's MAC address. Thus, mobility removes acl's flow deny rules related to host h2.

Next, the attacker host h1 attempts again to connect to the target host h2, and that causes the host provider to assume that host h1 has moved to its original location and thus triggers a HOST_MOVED event type. Prior to patching the vulnerability, acl did not check for the HOST_MOVED event type and took no action to reinstall the former flow deny rules. Another app, such as fwd, then installs flow allow rules from the attacker host h1 to the target host h2.

3) Results and implications: We wrote an exploit that performed the attack and were able to demonstrate that messages could be sent from the attacker to the target. Although the attack assumed that the attacker controlled two hosts on different connection points, an attacker who initially controls only one host could use the previous exploit in Section VII-A to compromise a second host so as to perform the attack in this section. Much like the exploit in Section VII-A, the increased access has significant consequences if our assumptions about the security of data plane access control are incorrect. For instance, if hosts h1 and h2 were segmented and isolated by policy (*e.g.,* to satisfy regulatory compliance requirements), then clever manipulation of host events can effectively bypass such protections.

4) Event flow graph: Figure 9 shows the partial event flow graph with the relevant code paths used by the attacker. The attack's first stage follows the path through the host mobility app, mobility, in the figure's center. The host mobility app responds to the HOST_MOVED event type and removes flow rules. The access control app, acl, does not handle the HOST_MOVED event, and thus the app does not install new flow rules. The attack's second stage succeeds as shown by the path on the right side of the figure.

In the analysis of acl, EVENTSCOPE produces a present context set, c_+, that includes mobility. The present context set indicates how the unhandled event type (*i.e.,* HOST_MOVED) is handled by other event handlers of the same event kind (*i.e.,* the HostEvent event kind). A practitioner would determine that mobility uses flow removal to produce undesirable effects. The absent context set, c_-, includes the forwarding app, fwd. A practitioner would determine that fwd uses flow rule installation to produce undesirable effects.

C. Other Vulnerabilities

In Table I, we summarize the remaining vulnerabilities that EVENTSCOPE discovered, grouped by app.

Vulnerabilities 3–5 (virtualbng): The virtual broadband network gateway app, virtualbng, maintains a relationship between a network's set of private IP addresses and public-facing IP addresses on the Internet [49]. The app also installs network intents, which get translated to new flow rules, to allow the network's hosts with private IP addresses to connect to the Internet. The app's host event listener handles the HOST_ADDED event type but does not handle the remaining three host event types. As a result, the app does not handle any state updates about the virtual gateways it has previously created if a host changes its information (*e.g.,* new location). A malicious host could spoof that host's identity, via a process similar to that described in Section VII-B2, to cause HOST_UPDATED or HOST_MOVED event types to be triggered. Furthermore, when a host is removed, the app does not asynchronously remove its intents (or, by extension, its flow rules) that it previously installed because it does not handle HOST_REMOVED event types.

Vulnerabilities 6–8 (mobility): The host mobility app, mobility, listens for host-related events and cleans up any related flow rules if a host has moved. Related work [20] has shown how the host mobility app in ONOS can be abused by hosts to force ONOS to reinstall flow rules and cause a control plane denial-of-service attack. Instead, we focus here on the absence of what event types mobility handles. The app's host event listener handles the HOST_MOVED event type (as expected) but does not handle the remaining three host event types. If mobility is expected by other apps to be responsible for cleaning up flow rules, then a host whose information has been updated (where updating would trigger a HOST_UPDATED event type), would not cause a flow removal and might lead to stale flow rules. If there is sufficient time between a moved host's removal from and addition back into the network, it may trigger a HOST_REMOVED event followed by a HOST_ADDED event. As mobility does not handle either event type, the expected flow removal by mobility would not occur.

Vulnerability 9 (vtn): The virtual tenant network app, vtn, provisions virtual networks as overlays over physical networks [50]. The app handles all of the host event types except for HOST_MOVED. For the host event types that

are handled, the app installs flow rules for added hosts (*i.e.,* HOST_ADDED), removes flow rules for removed hosts (*i.e.,* HOST_REMOVED), and installs and removes flow rules for any host that has changed its properties but not moved (*i.e.,* HOST_UPDATED). A host that moves (*i.e.,* HOST_MOVED) would not have any actions taken by the app; as a result, flow rules would not be reinstalled, and denial of service could occur.

Vulnerabilities 10–11 (evpnopenflow): The Ethernet VPN app, evpnopenflow, uses OpenFlow to install MPLS-labeled overlay routes for virtual private networks [47]. The app's host event listener handles the HOST_ADDED and HOST_REMOVED event types, which call functions that are responsible for finding routable paths, installing flow rules, and removing flow rules. The app does not handle hosts moving (*i.e.,* HOST_MOVED) or being updated (*i.e.,* HOST_UPDATED), and that could cause denial of service to such hosts if old flow rules are not removed and new flow rules are installed.

Vulnerabilities 12–14 (p4tutorial): The P4 tutorial app, p4tutorial, is a proof-of-concept app that demonstrates P4's programmable data plane capabilities. The app's host event listener handles the HOST_ADDED event type only. Like virtualbng, p4tutorial's lack of handling of other host event types leaves it susceptible to denial-of-service vulnerabilities and failure to remove flow rules.

VIII. Discussion

A. SDN Design Concerns

1) App composability: We found that some apps, which we term "helper apps," were designed to perform functionality on behalf of other apps currently running. One helper app, mobility, removes flow rules when hosts move within the network. However, as we noted with respect to our exploit in Section VII-B, if an app's design does not account for helper apps that are taking actions on its behalf, then the combination of apps may introduce vulnerabilities that arise from a lack of coordinated responsibility. That suggests a need for stronger integration testing among apps; EventScope is useful in identifying the subsets of apps that may interact.

2) Update semantics: We found that ONOS event kinds often had representations in their event types for updates (*i.e.,* *_UPDATED, *_CHANGED, or *_MOVED). While some apps handled the respective "addition" or "removal" event types, they did not handle the respective "updated" event type (*e.g.,* the odtn app for LINK_UPDATED). Apps that did handle update event types often did so by first calling a removal method, followed by an addition method; for instance, the vtn app handles HOST_UPDATED by calling its `onHostVanished()` and `onHostDetected()` methods consecutively. The lack of uniform update event-type handling across apps suggests that update handling is a useful place to identify vulnerabilities.

3) Host migration: Although host migration hijacking is a known problem [14], [22], [57], [26], we found that ONOS v1.14.0 and earlier versions do not provide any protections against the broader class of adversarial host-generated data plane input. That suggests a strong cross-plane attack vector, and EventScope's event flow graph can show the extent to which the control plane's control flow can be altered.

4) Event abstraction: While EventScope's discovered vulnerabilities do relate to host movement, such vulnerabilities differ from the host migration vulnerabilities discovered in related work [14], [22], [57], [26]. Those previously known vulnerabilities specifically use incoming data plane packets to target the host migration service. In contrast, EventScope's discovered vulnerabilities occur one abstraction layer higher: the host migration service declares that a host has moved, and other apps attempt to update their own states to account for such movement. EventScope's discovered vulnerabilities could occur as a result of benign host migration. For example, the acl app relies on a host migration service event (*i.e.,* HostEvent) instead of relying directly on data plane packets because the semantic notion of host migration is a useful abstraction for other apps, too. We believe that future apps will likely follow a similar trend of using abstracted events. One of our goals is to make event propagation more understandable for practitioners and developers. In that context, we believe that EventScope's discovered vulnerabilities are distinct from and complementary to the host migration vulnerabilities found in related work.

5) Other controllers: Much like ONOS's packet processor, Floodlight's [17] processing chains allow for specific execution ordering. ONOS contains a more sophisticated, extensive, and distributed event-driven architecture than Floodlight, and we opted to evaluate the more sophisticated architecture. ONOS also contains event processing that does not specify ordering, which is the case for the majority of ONOS event kinds (*i.e.,* all non-packet events). Although the event flow graph captures the ordering of different events (*e.g.,* a packet event that subsequently triggers a host event), the graph does not capture the processing order within an event (*e.g.,* the packet event goes to app X, then app Y).

B. Limitations

EventScope cannot establish the absence of vulnerabilities. NICE [11] shows that a large state space search is needed to reason about the absence of vulnerabilities, but such state does not scale beyond simple apps and controllers. EventScope lets developers and practitioners understand complex app interactions using a scalable approach.

To help practitioners identify unsafe operating conditions, EventScope can generate contexts under which certain combinations of apps may manifest a vulnerability; however, EventScope does not generate exploits. Automated exploit generation [3] is an ongoing research area, and we consider automated SDN exploit generation to be future work.

We believe that the event flow graph data structure has applicability beyond the identification of missing event vulnerabilities. For instance, concurrent event processing can be represented in an event flow graph by two paths with the same start and end nodes. Such path structures may indicate race conditions, and the event flow graph could be well-suited to identifying where these occur. However, we believe that that, and other possible applications, are complex research questions in their own right, and we leave them as future work.

IX. Related Work

SDN security: Cross-plane attacks have been studied in specific contexts. Yoon *et al.* [70] refer to these attacks as control plane remote attacks for network-view manipulation. SPHINX [14], TOPOGUARD [22], TOPOGUARD+ [57], and SECUREBINDER [26] reveal the lack of protection against link fabrication attacks and host location hijacking. However, none of the four systems analyze the extent to which the untrusted data plane inputs propagate via events to other components in the controller, and such analysis is necessary for cases where apps' competing behaviors create vulnerabilities.

CONGUARD [66] identifies time-of-check-to-time-of-use race conditions in SDN controllers and provides a generalized model of control plane happens-before relations, but the generalized semantics do not account for more sophisticated app semantics whose incomplete event handling can be exploited.

INDAGO [32] and SHIELD [33] use static analysis to analyze SDN apps and summarize their API use. INDAGO proposes machine learning techniques to determine whether an app is malicious or benign based on its sources and sinks from API call use. Given that benign apps can be co-opted by other apps as confused deputies [59], we find the distinction of malicious and benign labeling to be irrelevant for EVENTSCOPE. Instead, EVENTSCOPE approaches the problem from a global event dependency view.

Event-driven architectures: We consider the SDN architecture *vis-à-vis* Android and Web browser extensions. SDN and Android differ based on the mechanisms by which data are passed and on how apps coordinate with each other [59]. Event-driven SDN relies on a central event dispatching mechanism over a limited set of network events, which implies that SDN apps must coordinate with each other to apply policies to and to enforce security over the shared data plane resource. Vulnerability tools and analyses for Android [2], [15], [35], [38], [54], [73], [72], [69] and browser extensions [5], [6], [12], [36] have focused primarily on preventing information leakage among apps or extensions rather than specifically on how unhandled events affect global control flow.

Vulnerability discovery: Livshits and Lam [37] secure Java programs from unchecked Web-based input vulnerabilities. We study the analogous SDN problem of untrusted data plane input and model our attacks using a two-stage model of initial injection and subsequent manipulation. Yang *et al.* [68] note the challenges for event-driven callbacks in Android, which we consider in our SDN component model. Monperrus and Mezini [41] study the use of missing method calls as indicators of deviant code, using an approach similar to that used for the unhandled event type problem. CHEX [38] identifies entry points in Android applications and uses "app splitting" to identify all code that is reachable from a given entry point. We adapted CHEX's notion of app splitting for use in building event flow graphs. Code property graphs [67] combine abstract syntax trees, control flow graphs, and program dependence graphs into a unified data structure for automated vulnerability discovery, but have scalability concerns.

Network debugging: Cross-plane and cross-app attacks can be tracked using causality tracking and data provenance approaches. PROVSDN [59] prevents cross-app poisoning attacks in real time using a provenance graph structure to enforce information flow control, and FORENGUARD [62] records previous causal relationships to identify root causes. Negative provenance [65], differential provenance [13], and meta provenance [64] have been proposed to explain why SDN routing events did not occur and to propose bug fixes, but such methods require either a history of traces or reference examples of "good" behavior; furthermore, analysis of SDN applications in those systems must be written in or translated into the Datalog language NDlog prior to analysis. The aforementioned systems record code paths that were taken rather than all potential code paths, which limits their effectiveness in identifying potential vulnerabilities ahead of time.

DELTA [34], BEADS [25], and STS [55] use fuzzing to generate data plane inputs, but the space of potential inputs is complex for large and complex event-driven controllers. NICE [11] models basic control plane semantics (*e.g.,* flow rule installation ordering) and uses the generated state space to perform concrete symbolic (*i.e.,* concolic) execution to find bugs; however, even for simple single apps, the approach does not scale well. VeriFlow [29] uses network correctness properties to prevent flow rules from being installed. However, such approaches require a formal statement about app behavior. Given that the checks occur in the southbound API, such tools do not identify the sources of vulnerabilities.

X. Conclusion

We have presented EVENTSCOPE, a vulnerability discovery tool for SDN that enables practitioners and developers to identify cross-plane event-based vulnerabilities by automatically analyzing controller apps' event use. EVENTSCOPE uses similarities among apps to find potential logic bugs where event types are not handled by apps. EVENTSCOPE uses an event flow graph, which abstracts relevant information about how events flow within the control plane, captures data-plane inputs as potential cross-plane attack vectors, and captures data-plane outputs as targets. We used EVENTSCOPE on ONOS to find and validate 14 new vulnerabilities.

Acknowledgment

The authors thank our shepherd, Qi Li, and the anonymous reviewers for their helpful comments, which improved this paper; the PERFORM and STS research groups at the University of Illinois for their advice and feedback; and Jenny Applequist for her editorial assistance. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1657534 and CNS-1750024.

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 113–126.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of ACM PLDI '14*, 2014.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proceedings of NDSS '11*, Feb. 2011.

[4] A. Azzouni, O. Braham, T. M. T. Nguyen, G. Pujolle, and R. Boutaba, "Fingerprinting OpenFlow controllers: The first step to attack an SDN control plane," in *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec 2016, pp. 1–6.

[5] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: Vetting browser extensions for security vulnerabilities," in *Proceedings of USENIX Security '10*, 2010.

[6] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting browsers from extension vulnerabilities," in *Proceedings of NDSS '10*, 2010.

[7] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an open, distributed SDN OS," in *Proceedings of ACM HotSDN '14*, 2014.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[9] P. Biondi. (2019) Scapy: Packet crafting for python2 and python3. [Online]. Available: https://scapy.net/

[10] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of ACM ICSE '11*, 2011.

[11] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Proceedings of USENIX NSDI '12*, 2012.

[12] N. Carlini, A. P. Felt, and D. Wagner, "An evaluation of the Google Chrome extension security architecture," in *Proceedings of USENIX Security '12*, 2012.

[13] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The good, the bad, and the differences: Better network diagnostics with differential provenance," in *Proceedings of ACM SIGCOMM '16*, 2016.

[14] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *Proceedings of NDSS '15*. Internet Society, Feb. 2015.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of USENIX OSDI '10*, 2010.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[17] Floodlight, 2019. [Online]. Available: http://www.projectfloodlight.org/

[18] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker, "Frenetic: A high-level language for OpenFlow networks," in *Proceedings of ACM PRESTO '10*, 2010.

[19] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[20] R. Hanmer, S. Liu, L. Jagadeesan, and M. R. Rahman, "Death by babble: Security and fault tolerance of distributed consensus in high-availability softwarized networks," in *Proceedings of IEEE NetSoft '19*, June 2019, pp. 266–270.

[21] Hewlett Packard Enterprise, 2019. [Online]. Available: https://techlibrary.hpe.com/si/en/networking/solutions/technology/sdn/portfolio.aspx/

[22] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proceedings of NDSS '15*, Feb. 2015.

[23] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of ACM PASTE '07*, 2007.

[24] G. Jeh and J. Widom, "SimRank: A measure of structural-context similarity," in *Proceedings of ACM KDD '02*, 2002.

[25] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "BEADS: automated attack discovery in OpenFlow-based SDN systems," in *Proceedings of RAID '17*, 2017.

[26] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *Proceedings of USENIX Security '17*, 2017.

[27] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of ICSE '13*, 2013.

[28] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 1990.

[29] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[30] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[31] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of ACM HotNets '10*, 2010.

[32] C. Lee, C. Yoon, S. Shin, and S. K. Cha, "INDAGO: A new framework for detecting malicious SDN applications," in *Proceedings of IEEE ICNP '18*, Sep. 2018.

[33] C. Lee and S. Shin, "SHIELD: An automated framework for static analysis of SDN applications," in *Proceedings of ACM SDN-NFV Security '16*, 2016.

[34] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proceedings of NDSS '17*, Feb. 2017.

[35] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proceedings of ICSE '15*, 2015.

[36] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome extensions: Threat analysis and countermeasures," in *Proceedings of NDSS '12*, 2012.

[37] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proceedings of USENIX Security '05*, 2005.

[38] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proceedings of ACM CCS '12*, 2012.

[39] 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.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[40] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of ACM SOSP '15*, 2015.

[41] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 7:1–7:25, Mar. 2013.

[42] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 217–230.

[43] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing software defined networks," in *Proceedings of NSDI '13*, 2013.

[44] B. Naveh. (2019) Jgrapht. [Online]. Available: https://jgrapht.org/

[45] Open Networking Foundation, "In action - ONOS." [Online]. Available: https://www.onosproject.org/in-action/

[46] ——. (2019) Github – opennetworkinglab/onos at 1.14.0. [Online]. Available: https://github.com/opennetworkinglab/onos/tree/onos-1.14

[47] ——. (2019) Overlay VPNs and Gluon. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Overlay+VPNs+and+Gluon

[48] ——. (2019) System components – ONOS. [Online]. Available: https://wiki.onosproject.org/display/ONOS/System+Components

[49] ——. (2019) Virtual BNG. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Virtual+BNG

[50] ——. (2019) Virtual network subsystem. [Online]. Available: https://wiki.onosproject.org/download/attachments/6357849/VirtualNetworkSubsystem.pdf

[51] OpenDaylight. (2019) Home - opendaylight. [Online]. Available: https://www.opendaylight.org/

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.

[53] PMD. (2019) PMD: An extensible cross-language static code analyzer. [Online]. Available: https://pmd.github.io/

[54] C. Qian, X. Luo, Y. Le, and G. Gu, "VulHunter: Toward discovering vulnerabilities in Android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan 2015.

[55] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "Troubleshooting blackbox SDN control software with minimal causal sequences," in *Proceedings of ACM SIGCOMM '14*, 2014.

[56] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of IEEE S&P '16*, May 2016, pp. 138–157.

[57] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, "Effective topology tampering attacks and defenses in software-defined networks," in *Proceedings of IEEE/IFIP DSN '18*, June 2018, pp. 374–385.

[58] J. Toman and D. Grossman, "Taming the static analysis beast," in *Proceedings of SNAPL '17*, ser. LIPIcs, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., vol. 71. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, pp. 18:1–18:14.

[59] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, "Cross-app poisoning in software-defined networking," in *Proceedings of ACM CCS '18*, 2018.

[60] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proceedings of CASCON '10*, 2010.

[61] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proceedings of ACM HotSDN '12*, 2012, pp. 43–48.

[62] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards fine-grained network security forensics and diagnosis in the SDN era," in *Proceedings of ACM CCS '18*, 2018.

[63] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *Proceedings of USENIX OSDI'12*, 2012.

[64] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated bug removal for software-defined networks," in *Proceedings of USENIX NSDI '17*, Mar. 2017.

[65] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," in *Proceedings of ACM SIGCOMM '14*, 2014.

[66] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN control plane," in *Proceedings of USENIX Security '17*, 2017.

[67] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of IEEE S&P '14*, 2014.

[68] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *Proceedings of ICSE '15*, 2015.

[69] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of ICSE '15*, 2015.

[70] C. Yoon, S. Lee, H. Kang, T. Park, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3514–3530, Dec. 2017.

[71] M. Zhang, J. Hou, Z. Zhang, W. Shi, B. Qin, and B. Liang, "Fine-grained fingerprinting threats to software-defined networks," in *Proceedings of IEEE Trustcom/BigDataSE/ICESS '17*, Aug 2017, pp. 128–135.

[72] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proceedings of ACM CCS '14*, 2014.

[73] M. Zhang and H. Yin, "AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications," in *Proceedings of NDSS '14*, 2014.

```
1  package org.onosproject.sampleApp;
2  public class SampleAppManager {
3    /* Internal variables */
4    protected HostService hostService;
5    protected PacketService packetService;
6    protected FlowRuleService flowRuleService;
7    private HostListener hostListener = new HL();
8    private PacketProcessor processor = new PP();
9    /* Activation and deactivation methods */
10   protected void activate() {
11     ...
12     hostService.addListener(hostListener);
13     packetService.addProcessor(processor, 0);
14   }
15   protected void deactivate() {
16     ...
17     packetService.removeProcessor(processor);
18     hostService.removeListener(hostListener);
19   }
20   /* Event listener(s) */
21   private class HL implements HostListener {
22     public void event(HostEvent event) {
23       switch (event.type()) {
24         case HOST_ADDED:
25           internalMethod1(event,...);
26         default:
27       }
28     }
29   }
30   /* Packet processor(s) */
31   private class PP implements PacketProcessor {
32     public void process(PacketContext context) {
33       ...
34       internalMethod2(...)
35     }
36   }
37   /* App internal methods (public or private) */
38   private void internalMethod1(Event event,...) {
39     ...
40     internalMethod2(...)
41   }
42   public void internalMethod2(...) {
43     ...
44     flowRuleService.applyFlowRules(...);
45   }
46 }
```

Fig. 10: Abbreviated code structure of an example ONOS network application, sampleApp, written in Java.

APPENDIX

A. ONOS Application Structure

1) App Components: We provide an example ONOS app with representative components. Figure 10 shows the representative code structure of an example application, sampleApp. sampleApp listens for host events and incoming data plane packets; based on such events, the app installs new flow rules. We highlight the key components of an ONOS app below.

- **Internal variables (lines 3–8):** Internal variables maintain the app's state, which includes references to data store objects and core controller services. In sampleApp, references to the host, packet, and flow rule services are created, along with the instantiations of the host (event) listener and the packet processor.
- **Activation and deactivation methods (lines 9–19):** The activation method is called once when the app is activated; similarly, the deactivation method is called once during deactivation. During activation and deactivation, the app registers and deregisters components that it needs, such as event listeners and packet pro-

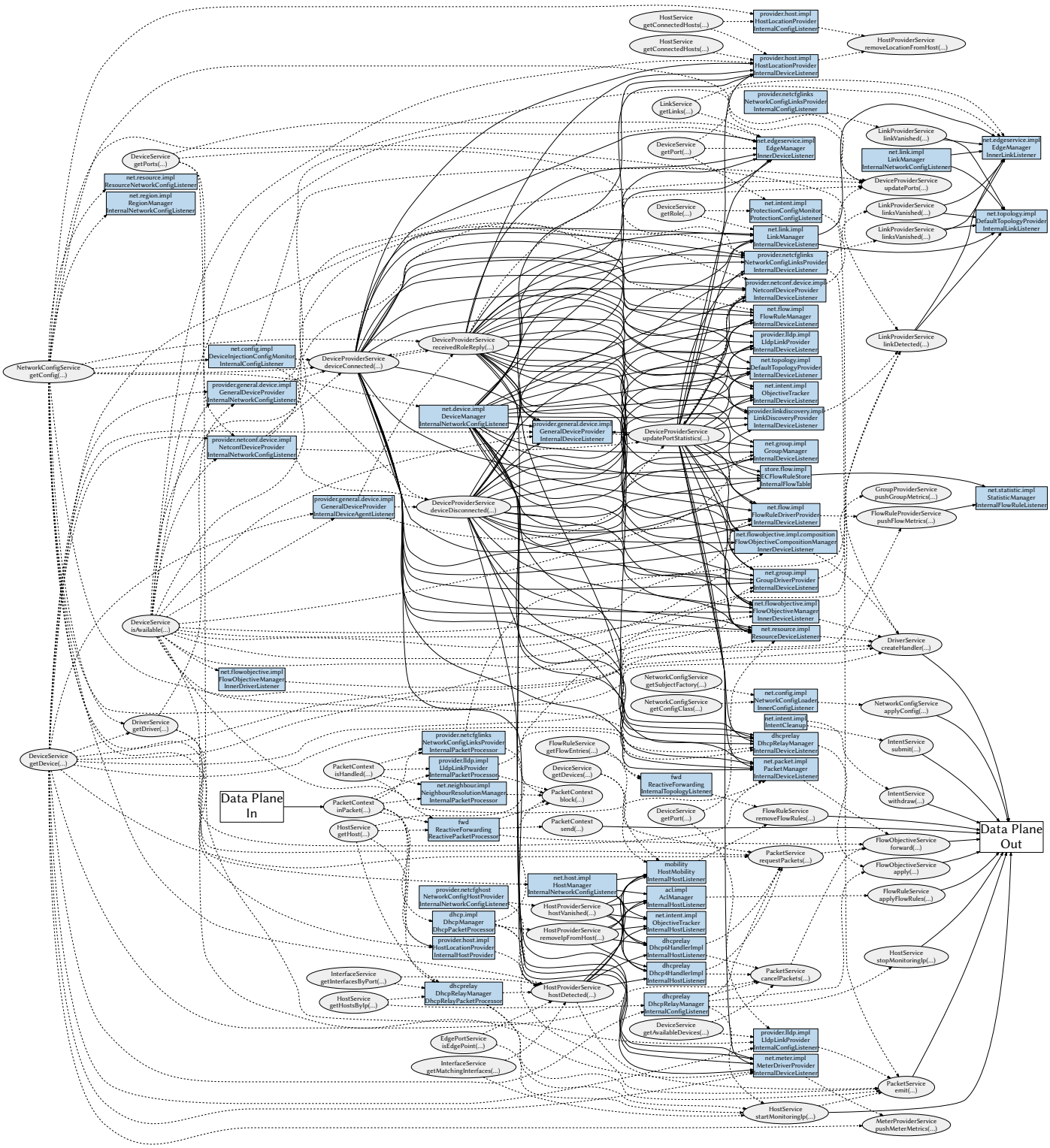


Fig. 11: Event flow graph of ONOS with core service components and several apps (*i.e.,* acl, fwd, mobility, dhcp, and dhcprelay). Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, bold edges represent event dispatches, and dashed edges represent API calls. (For simplicity, event types are reduced to a single edge of the event type's respective event kind.)

cessors. In sampleApp, the host event listener and packet processor are registered and deregistered.

- **Event listeners (lines 20–29):** Event listeners listen for an event kind of interest and take further action, often based on the event type. Event listeners may call other methods within the app to perform a desired functionality. In sampleApp, the host event listener executes `event()` when it receives a HostEvent (line 22). It handles the HOST_ADDED type by calling the internal method `internalMethod1()` (line 25). Note that all other HostEvent event types (*e.g.,* HOST_REMOVED) are not handled.
- **Packet processors (lines 30–37):** Packet processors function much like to event listeners by listening for incoming data plane packets and taking appropriate actions. In sampleApp, the packet processor executes `process()` when it receives a packet (line 32) and subsequently after execution calls the internal method `internalMethod2()` (line 34).
- **App internal methods (lines 38–45):** App internal methods handle the main functionality of the app. They may read from core services (*i.e.,* API read calls), write to core services (*i.e.,* API write calls), or dispatch new events. In sampleApp, `internalMethod1()` calls `internalMethod2()`. New flow rules are generated as a result of the calling of `internalMethod2()` (line 44).

2) App Analysis: We explain how sampleApp would be analyzed within EVENTSCOPE.

a) Event use: Based on the event listener that is implemented in sampleApp, we see that the HostEvent event is handled. For simplicity, $E_K = \{\text{HostEvent}\}$ and $E_T = \{\text{HOST\_ADDED}, \text{HOST\_REMOVED}, \text{HOST\_MOVED}, \text{HOST\_UPDATED}\}$. Because sampleApp handles only the HOST_ADDED event type, its corresponding row in the event use matrix, M, would be $M[\text{sampleApp}] = [\texttt{true}, \texttt{false}, \texttt{false}, \texttt{false}]$. Next, sampleApp's event types would be compared with respect to all other apps to determine if the 3 remaining event types are candidates.

b) Event flow: Given that apps' event listeners and packet processors drive the main app functionality, EVENTSCOPE focuses on these methods and ignores the activation and deactivation methods. We mark the host event listener `event()` method (line 23) and the packet processor `process()` method (line 32) as entry points for the event flow graph generation. Each entry point is represented as a node in the event flow graph, $\mathcal{G}$. We note that `flowRuleService.applyFlowRules()` is an API write method, so it would also be marked as an entry point.

For the host event listener, the resulting call graph contains the path `event()` → `internalMethod1()` → `internalMethod2()` → `applyFlowRules()`, so we add an outgoing edge from the host event listener node to the flow rule API call node in $\mathcal{G}$. For the packet processor, the resulting call graph contains the path `process()` → `internalMethod2()` → `applyFlowRules()`, so we add a similar edge from the packet processor node to the flow rule API call node. As the host event listener handles only 1 event type, we add 1 edge from each host event dispatcher node (assumed to have been dispatched from other activated controller code) to sampleApp's host event listener in $\mathcal{G}$.

Finally, as the packet processor receives incoming data plane input, we add an edge from `DPIn` to the packet processor in $\mathcal{G}$. As the host event listener and packet processor add flow rules, we add edges from each to `DPOut` in $\mathcal{G}$.

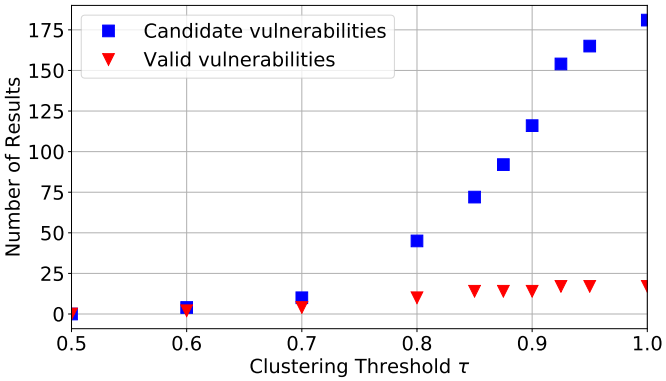


Fig. 12: ONOS apps' candidate and valid vulnerabilities as a function of clustering threshold τ (using SimRank [24]).

B. ONOS Event Flow Graph Example

Figure 11 shows the ONOS event flow graph with the controller's core services, the access control app (acl), the reactive forwarding app (fwd), the host mobility app (mobility), and the DHCP apps (dhcp and dhcprelay).

We start from the `Data Plane In` node on the left side of the figure, where the `inPacket()` API read call receives incoming data plane packets. Such packets are read by several packet processors: the neighborhood service's InternalPacketProcessor, the reactive forwarding app's ReactivePacketProcessor, the LLDP link provider's InternalPacketProcessor, the DHCP apps' DhcpPacket Processor and DhcpRelayPacketProcessor, and the host location provider's InternalHostProvider.

We follow paths from left to right to understand how those packet processors cause subsequent API calls and event dispatches. For instance, the dhcprelay app calls the HostProviderService's `hostDetected()` API call. The `hostDetected()` API call will dispatch a HostEvent event that gets received by the dhcprelay app's InternalHostListener event listener. That event listener calls the PacketService's `cancelPackets()` API call, which subsequently calls the FlowObjective Service's `forward()` API call. The `forward()` API call causes a data plane effect.

C. Number of Clusters and Detection Rate

Figure 12 shows the effect of choosing different values for the event use clustering threshold τ (*i.e.,* changing the number of clusters) on the detection rate for the number of candidate vulnerabilities (Section IV-A) and valid vulnerabilities (Section V-B) for ONOS v1.14.0 [46]. We note an inflection point of candidate vulnerabilities near $\tau = 0.90$, which is the threshold that we used throughout our evaluation.