Cross-App Poisoning in Software-Defined Networking

Benjamin E. Ujcich
University of Illinois at Urbana-Champaign

Samuel Jero
MIT Lincoln Laboratory

Anne Edmundson
Princeton University

Qi Wang
University of Illinois at Urbana-Champaign

Richard Skowyra
MIT Lincoln Laboratory

James Landry
MIT Lincoln Laboratory

Adam Bates
University of Illinois at Urbana-Champaign

William H. Sanders
University of Illinois at Urbana-Champaign

Cristina Nita-Rotaru
Northeastern University

Hamed Okhravi
MIT Lincoln Laboratory

ABSTRACT

Software-defined networking (SDN) continues to grow in popularity because of its programmable and extensible control plane realized through network applications (apps). However, apps introduce significant security challenges that can systemically disrupt network operations, since apps must access or modify data in a shared control plane state. If our understanding of how such data propagate within the control plane is inadequate, apps can co-opt other apps, causing them to poison the control plane's integrity.

We present a class of SDN control plane integrity attacks that we call *cross-app poisoning (CAP)*, in which an unprivileged app manipulates the shared control plane state to trick a privileged app into taking actions on its behalf. We demonstrate how role-based access control (RBAC) schemes are insufficient for preventing such attacks because they neither track information flow nor enforce information flow control (IFC). We also present a defense, ProvSDN, that uses data provenance to track information flow and serves as an online reference monitor to prevent CAP attacks. We implement ProvSDN on the ONOS SDN controller and demonstrate that information flow can be tracked with low-latency overheads.

CCS CONCEPTS

• **Security and privacy** → *Access control*; *Information flow control*; *Information accountability and usage control*; • **Networks** → *Programmable networks*;

KEYWORDS

software-defined networking; data provenance; information flow control; network operating system

*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.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243759

ACM Reference Format:
Benjamin E. Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H. Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. 2018. Cross-App Poisoning in Software-Defined Networking. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243759

1 INTRODUCTION

Software-defined networking (SDN) has emerged as a flexible architecture for programmable networks, with deployments spanning from enterprise data centers to cloud computing and virtualized environments, among others [33]. The rapid growth and potential value[1] of SDN stems from the need in industry and the research community for dynamic, agile, and programmable networks. Driving the popularity of SDN is the use of modular and composable *network applications* (or *apps*) that extend the capabilities of the logically centralized control plane. Networks that would formerly have required monolithic and proprietary software or complex middlebox deployment can now be addressed by the larger developer community through the use of application program interfaces (APIs) and even third-party app stores for practitioners [31].

While apps add value in ways that would have been difficult or impractical before, the burgeoning SDN app ecosystem introduces significant control plane security challenges. The SDN architecture arguably involves a larger attack surface than traditional networks, because malicious apps can disrupt network operations systemically and significantly [14, 37, 42, 73]. A recent article notes that "attacks against SDN controllers and the introduction of malicious controller apps are probably the most severe threats to SDN," and that the situation is further complicated by dynamic configurations that make it impossible for "defenders to tell whether the current or past configuration is intended or correct" [14].

To date, defenses that limit the SDN attack surface have included app sandboxing [75], TLS-enabled APIs [61, 64, 68], API abuse prevention [65, 78], and role-based access control (RBAC) for apps [68, 89], among others. Although these mechanisms improve control plane security, we posit that they are not sufficient for mitigating information flow attacks within the control plane.

[1] A 2016 forecast by the International Data Corporation predicts that the SDN market will be valued up to 12.5 billion USD by 2020, with network applications accounting for 3.5 billion USD of that market [33].

In order to function properly, apps necessarily require access to and/or modification of the SDN control plane state, which includes data stores and control plane messages. This "shared" state design among apps creates new attack vectors for integrity attacks. For instance, trusted or system-critical apps may unintentionally use data generated by untrusted or malicious apps [42], leading to a "confused deputy" problem [30]. To date, the SDN security literature has not systematically considered the class of integrity attacks that leverage information flow within the control plane, leaving SDN controllers that implement this shared state design vulnerable.

While RBAC-based systems can limit the attack surface by preventing access to shared data structures based on assignment of permissions to roles and subjects, RBAC alone is not sufficient for preventing attacks against the integrity of the shared SDN control plane state, because RBAC does not track how data are used after authorization [72]. Consider the scenario in which an SDN controller provides host and flow rule services among its core functionalities. Suppose an adversary has compromised a host-tracking app that, as part of the app's normal functionality, has permission to write to the host data store, but does not have permission to write flow rules. A second app performing routing has permission to read the host store and also to read and write flow rules. As part of its functionality, the routing app ensures that all hosts can be routed correctly, and it modifies flow rules as needed. Now suppose that the adversary modifies a host location in the host data store to point to a host that it has compromised. The routing app detects this change and rewrites flow rules to reflect the new location. Without being granted permission, the host-tracking app in this example has succeeded in effectively bypassing the RBAC-based system by having the routing app modify the network's flow rules on the host-tracking app's behalf.

In this paper, we analyze information flow within SDN control planes in order to consider the vulnerabilities inherent in the SDN architecture's design, the attack surface that the design introduces, and possible mitigation strategies based on information flow control (IFC) to ensure the control plane's integrity. We introduce and formalize a class of information flow attacks in the SDN control plane that we call *cross-app poisoning (CAP)*, in which a lesser-privileged app can co-opt another app so that the compromised app takes privileged actions on behalf of the attacking app. We have modeled the attack surface with a *cross-app information flow graph* that maps relations among apps through the shared control plane state and granted permissions.

Using the 64 apps included with the popular ONOS SDN controller [61] as a representative case study, we generated a least-privilege reference security policy using API-level permissions from the RBAC-based Security-Mode ONOS variant [89]. With our API-level RBAC policy, we generated and analyzed a cross-app information flow graph to identify opportunities for CAP attacks based on the overlapping permissions granted to shared data objects. To validate our results, we generated data flow graphs of ONOS apps to identify a set of *CAP gadgets* that can be used to instigate CAP attacks, and, through a proof-of-concept attack, we demonstrated the existence of this vulnerability even among a curated set of apps.

To detect and prevent such attacks in real-time according to a desired IFC policy, we introduce our defense, ProvSDN: an online reference monitor for the SDN control plane that leverages a data provenance approach to track and record information flow in the control plane across app requests. ProvSDN intercepts API requests, tracks how the control plane state is subsequently used, and stores such metadata in a provenance graph that efficiently queries past history while also recording the control plane's history. For our implementation, we instrumented ONOS with ProvSDN and found that ProvSDN can, on average, enforce IFC by imposing an additional 17.9 ms on a new flow rule instantiation, suggesting that ProvSDN can be practical in security-conscious settings.

In summary, our main contributions are:

(1) The identification of the **IFC integrity problem in SDN**, *i.e.,* **cross-app poisoning (CAP)**. We demonstrate that malicious apps can utilize a lack of information flow protections to poison the control plane's state and escalate privilege.
(2) A systematic approach to identification of **CAP attack vulnerabilities**, given a specified RBAC policy, by modeling the SDN control plane's allowed information flows.
(3) A defense against CAP attacks, **ProvSDN**, that uses data provenance for detection and prevention of CAP attacks by enforcing IFC policies online in real-time.
(4) An **implementation and evaluation** of CAP attacks and ProvSDN with the ONOS controller.

This paper is organized as follows. In Section 2, we outline the threat model depicting our attacker's capabilities and goals. In Section 3, we provide background and an overview of SDN, how apps use the SDN control plane, and information flow challenges in the SDN control plane. In Section 4, we present our methodology for detecting CAP attacks. In Section 5, we show CAP attacks' existence using Security-Mode ONOS as a case study. In Section 6, we outline IFC policies to counteract CAP attacks. In Section 7, we present the design, implementation, and evaluation of our defense, ProvSDN. In Section 8, we discuss challenges and design trade-offs, and in Section 9, we discuss related work. We conclude in Section 10.

2 THREAT MODEL

We assume that the SDN controller is trusted and adequately secured but that it may provide services to, and be co-opted by, malicious SDN apps. We assume that apps may originate from third parties[2], such as app stores[3], and are thus untrusted and potentially malicious. Although network and security practitioners will use best practices and due diligence in vetting apps before deployment (*e.g.,* verifying that an app has been signed by a trusted developer), compiled apps without available source code are "black boxes" whose behavior the practitioners may not entirely understand and whose code may be vulnerable to compromise in unexpected ways.

We assume that an attacker controls a malicious app that has least-privileged RBAC permissions. The attacker's goal is to cause arbitrary flow rules to be installed so as to affect data plane operations, despite not having the permission to do so. SDN controllers

[2]For instance, ONOS allows third-party app developers to submit apps to be included in the controller's repository. ONOS and its apps are currently used by transport network providers and have been incorporated into commercial products developed by Huawei and Samsung, among others [62]. As of August 2018, ONOS has also been issued 12 CVE entries, including arbitrary apps being loaded into the controller [63].
[3]Aruba Networks, a subsidiary of Hewlett-Packard Enterprise, maintains an "SDN app store" for the HP SDN controller [31]. As of August 2018, the app store contained 12 apps from third-party developers and 13 apps from "Aruba Technology partners."

that do not implement RBAC make it trivially easy for apps to modify and poison data that other apps use. Lee *et al.* [41] cite the lack of access control in SDN controllers as the cause of several types of inter-app attacks, such as internal storage misuse, application eviction, and event listener unsubscription. Our goal is to understand these kinds of attacks *even after RBAC has been applied*, particularly under a conservative least-privileges model whose privileges are minimally necessary for app functionality.

Not all cross-app information exchanges are malicious in intent, and some may be desirable based on a given situation. However, *current SDN controllers do not allow for the ability to distinguish between benign and malicious cross-app information exchanges* because they do not track control plane information flow. A successful defender must be able to make this distinction.

We further assume that apps have principal identities and that the controller ensures that one app cannot forge actions such that they appear to have been taken by another app. That policy can be enforced using a public key infrastructure (PKI) for authentication [37], and several controllers (*e.g.,* [68]) already do so.

3 BACKGROUND AND OVERVIEW

We provide a brief overview of the SDN architecture's important components, how apps interact with the SDN control plane, the control plane's information flow challenges, and our main contributions in solving such challenges.

3.1 SDN Architecture

SDN decouples decisions on how traffic ought to be forwarded (*i.e.,* the control plane) from the forwarding itself (*i.e.,* the data plane). SDN centralizes this control in a logically centralized controller, exposes APIs to apps, and abstracts the lower-level details of network forwarding devices (*e.g.,* switches) [38]. Figure 1a shows a representative SDN architecture with the controller, apps, forwarding devices, and end hosts. Next, we highlight some relevant background on the controller and control plane APIs.

Controller. The controller acts as a "network operating system" to coordinate concurrent applications, to provision resources, and to implement security or network policies [38]. Several controller frameworks exist, such as Floodlight [20], Ryu [71], Open Network Operating System (ONOS) [61], and OpenDaylight (ODL) [64]. Special-purpose controllers for secure environments include SE-Floodlight [68], Rosemary [75], and Security-Mode ONOS [89]. The controller maintains data stores that collectively serve as a "network information base" for abstractions of the network's topology, flow entries, and end hosts, among others. The core methods provide services that add, modify, or remove data from data stores.

Northbound API. There is no standard controller-to-application interface or *northbound API* (*NB API*) among all controller frameworks, and each framework may establish different boundaries between the core functionalities and extensible apps. Apps can be implemented in two ways: as "internal" modules within the controller (represented by the dashed box in Figure 1a) or as separate "external" processes decoupled from the controller. For example, ONOS uses the OSGi framework in Java to manage internal app modules and states. ONOS, ODL, and Floodlight, among others, include a RESTful API for external apps.

Southbound API. SDN controllers also interact with network forwarding devices to disseminate rules and to collect data plane statistics. One popular standard protocol between controllers and switches (*i.e.,* the *southbound API*) is OpenFlow [46]. OpenFlow configures switches' forwarding behavior through flow tables, where each flow table consists of flow entries that match attributes of incoming data plane packets and assigns data plane forwarding actions. The protocol includes messages to send data plane packets to and from the controller (*i.e.,* `PacketIn`, `PacketOut`), to modify forwarding behavior (*i.e.,* `FlowMod`), and to request and receive flow entry statistics (*i.e.,* `StatsRequest`, `StatsReply`), among others.

SDN Control Plane App Interactions. Apps interact with the shared SDN control plane state via *service API calls* and *event callbacks*. These mechanisms are independent of whether apps are internal or external to the controller. With service API calls, an app can read from or write to one of the controller's data stores via a corresponding service and the service's methods. As shown in Figure 1b, apps use the host service's read() and write() methods to interact with the underlying host data store. With event callbacks, an app registers itself with the controller to receive events of interest as they occur. As shown in Figure 1c, all apps have registered to receive data plane events from a data plane event listener. Subsequent events may be generated as a result of the first event.

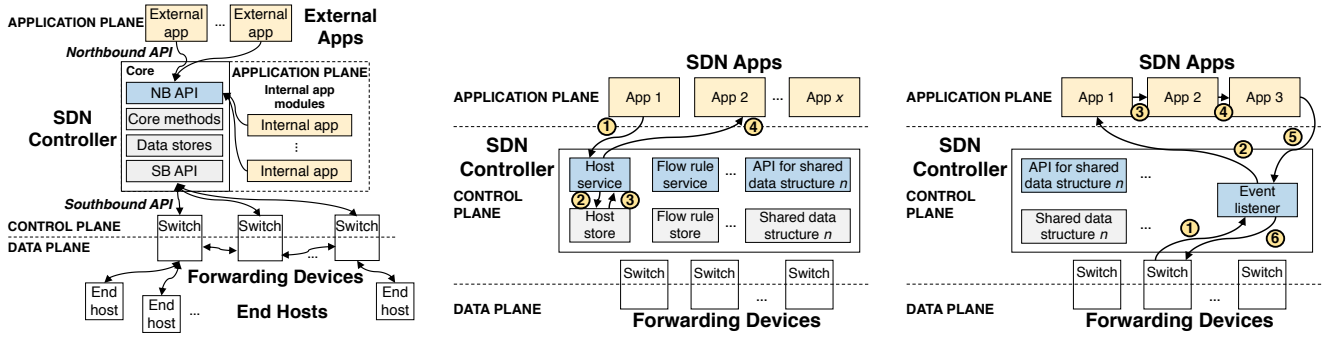
3.2 Information Flow Models for Integrity

Information flow concerns the extent to which data propagate throughout a system (*i.e.,* the SDN control plane) and influence other data. Information flow control (IFC) determines the ability of data to flow based on policy so as to enforce an "end-to-end" secure design by tracking propagation [72]. Pasquier *et al.* [66] provide an overview of classical information flow models. Among them is one proposed by Biba [10], who proposed a "no read down, no write up" integrity policy. In that model, subjects are assigned to one of several hierarchical integrity classes. Information can flow from a sender subject to a receiver subject if the sender's integrity class is at least as high as that of the sender, which implies that low-integrity information cannot reach high-integrity subjects. Myers and Liskov [53] relax the hierarchical assumptions by proposing a system of integrity tags and labels assigned to subjects.

3.3 SDN Control Plane Information Flow Challenges

Given that apps can interact with each other through the shared SDN control plane state, an ideal SDN controller must be able to capture the resulting information flow and enforce access control policies based on it. In considering the "network operating system" concept for SDN, we next highlight how current state-of-the-art SDN controller designs fall short with respect to information flow and IFC, and how we approach such challenges.

3.3.1 Lack of well-defined application isolation and enforcement as applied to shared control plane state. Some controllers, such as Rosemary [75], sandbox each app's resources (*e.g.,* memory and CPU usage) and use RBAC to allow apps or prevent them from



(a) Architecture. SDN separates the data and control planes to logically centralize network control. The application plane modularly extends the control plane functionality. Apps can reside either as modules within the controller or as external processes.

(b) Service API interactions. 1: App 1 calls one of the host service's write() methods to insert a new host. 2: The host service adds the object to its own data store. 3: App 2 calls one of the host service's read() methods (not shown), and the service queries the store. 4: The host service returns the object to app 2.

(c) Event callback interactions. 1: A switch notifies the controller about an event. 2: The event listener sends the event to the first registered app. 3 and 4: Additional registered apps receive the event. 5: The last app optionally returns an event. 6: The event is actuated (*e.g.,* in the data plane).

Figure 1: SDN architecture overview and app interactions via service APIs and event callbacks.

accessing parts of the SDN control plane state, in a manner analogous to resource sharing and file permissions in operating systems, respectively. However, RBAC is limiting in practice because it does not enforce certain usage of data after authorization [72]. Apps can bypass RBAC policies if they cleverly influence other apps to take actions on their behalf as "confused deputies."

Our contributions. We formalize this IFC integrity problem, under the name *cross-app poisoning (CAP)*, in Section 4, and demonstrate its consequences through an attack evaluation in Section 5.

3.3.2 Lack of insight into information flow within the control plane. A security practitioner might want to understand the control plane's information flow to evaluate the extent to which apps' information sharing should or should not be allowed. However, to date, there are no SDN controller logging mechanisms that explicitly and easily capture the relationships among the various ways data have been used or generated. Practitioners must manually reconstruct and infer possible scenarios by inspecting log files of varying verbosity. That makes it difficult or impossible to reason about prior network state [14, 37] or to quickly narrow down and attribute blame to specific apps when something goes wrong [48]. This lack of insight could mislead practitioners into incorrect conclusions when they investigate their systems.

Our contributions. In Section 4, we describe how to use a cross-app information flow graph to better understand the attack surface. In Section 7, we show how data provenance can provide insight into enforcement and recording of control plane activities.

4 CROSS-APP POISONING

We now introduce *cross-app poisoning (CAP)* as the IFC integrity problem for SDN. Informally, a *CAP attack* is any attack in which an app that does not have permission to take some action co-opts apps that do have such permissions by poisoning the other apps' view of data in the shared control plane state so that they take unintended or malicious actions on the first app's behalf.

To systematically identify CAP attacks, we model how apps are allowed to use and generate data based on how permissions are granted (Sections 4.1–4.3), and we overlay this model with apps' actual data flows (Section 4.4). While individual examples of CAP attacks have been considered in the SDN security literature (*e.g.,* [42]), we are (to the best of our knowledge) the first to systematically study this class of attacks, which cannot be prevented by the existing defenses in SDN, such as RBAC or app sandboxing.

4.1 RBAC Policy Model

We start with the current state-of-the-art in SDN secure controller design by considering an RBAC model as a basis for formalizing CAP attacks. Our model for specifying RBAC policies is denoted by $\mathcal{R} = (A, R, O, P_R, P_W, P, m_{AR}, m_{RP}, m_{PO})$ and consists of:

- A set of apps, denoted by $A = \{a_1, a_2, \ldots, a_x\}$, that comprise the apps in the SDN application plane.
- A set of roles, denoted by $R = \{r_1, r_2, \ldots, r_y\}$.
- A set of objects, denoted by $O = \{o_1, o_2, \ldots, o_z\}$, that comprise the data in the shared SDN control plane state.
- A set of read permissions, denoted by P_R, that make it possible to access or read from objects.
- A set of write permissions, denoted by P_W, that make it possible to write, modify, or delete objects.
- A union of all permissions, denoted by $P = P_R \cup P_W$.
- A mapping of apps to roles, denoted by $m_{AR} \subseteq A \times R$.
- A mapping of roles to permissions, denoted by $m_{RP} \subseteq R \times P$.
- A mapping of permissions to objects in the shared SDN control plane state, denoted by $m_{PO} \subseteq P \times O$.

Our RBAC model is flexible enough to be applied to several existing controllers. For instance, Security-Mode ONOS specifies objects and permissions at the API granularity (*e.g.,* read flow tables), whereas

Algorithm 1 Cross-App Information Flow Graph Generation

Input: RBAC policy $\mathcal{R}$
Output: cross-app information flow graph $\mathcal{G}$
Initialize:
$(A, R, O, P_R, P_W, P, m_{AR}, m_{RP}, m_{PO}) \leftarrow \mathcal{R}$
$\mathcal{V} \leftarrow A \cup O$
$\mathcal{E} \leftarrow \{\}$
1: **for each** $(a_i, r_i) \in m_{AR}$ **do**
2: **for each** $\left(r_j, p_j\right) \in m_{RP}$ such that $r_j = r_i$ **do**
3: **for each** $(p_k, o_k) \in m_{PO}$ such that $p_k = p_j$ **do**
4: **if** $p_k \in P_R$ **then**
5: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(o_k, a_i)\}$
6: **if** $p_k \in P_W$ **then**
7: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(a_i, o_k)\}$
8: $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E})$

SDNShield [85] specifies objects at the sub-API granularity (*e.g.,* read flow tables with a specific IP prefix).

4.2 Cross-App Information Flow Graph

Given a model and policies encapsulated in $\mathcal{R}$, we can convert $\mathcal{R}$ into a representation by which we can reason about potential data or information flow across the shared SDN control plane state. A *cross-app information flow graph*, denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a directed graph that encapsulates the relations among apps, objects in the shared SDN control plane state, and the permissions granted to apps to read and write objects. Our design is influenced by the "take–grant" protection model proposed by Lipton and Snyder [43].

Algorithm 1 shows the generation process, which uses a system modeled with an RBAC policy as input and a cross-app information flow graph as output. The algorithm initializes the components from $\mathcal{R}$ as well as the graph's nodes $\mathcal{V}$ as the union of apps A and objects O. Lines 1–3 iterate through RBAC maps so as to map each app–object pair. Each app–object pair may have zero or more permissions associated with it. For a read permission (lines 4–5), an edge is added to $\mathcal{E}$ from the object o_k to the app a_i. For a write permission (lines 6–7), an edge is added to $\mathcal{E}$ from the app a_i to the object o_k. Thus, the directions of the cross-app information flow graph's edges have semantic meaning based on reads and writes.

4.3 Cross-App Attack Vectors

Given a cross-app information flow graph $\mathcal{G}$, we can formally and precisely define CAP attacks in terms of paths in $\mathcal{G}$. We represent a *cross-app attack vector*, denoted by C_v, as a path in $\mathcal{G}$ such that the path's starting node is an app, the path's ending node is an object, the path length is greater than or equal to 3, and the path length is odd. (A path length of 1 represents what an app already has permission to do.) Based on the structure of $\mathcal{G}$ produced from Algorithm 1, the path nodes alternate between apps and objects. We define $C_v\ (\mathcal{G}) = \langle a_0, o_1, a_2 \ldots, a_{n-1}, o_n \rangle \mid n \geq 3; n$ is odd.

Intuitively, we can see that a path between an app and an object in $\mathcal{G}$ marks the existence of a potential attack vector. Any intermediate apps in a given C_v path are the apps that app a_0 can co-opt using permissions that a_0 itself does not possess. Similarly, any intermediate objects in a given C_v path are the objects in the shared SDN control plane state used to carry out the attack. For the trivial case in which systems do not implement any access control, $\mathcal{G}$ can be represented as a complete directed graph in which all apps can read from or write to all objects.

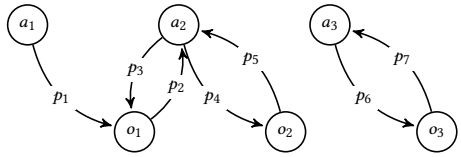


Figure 2: Example of a cross-app information flow graph $\mathcal{G}$ with a cross-app attack vector $C_1 = \langle a_1, o_1, a_2, o_2 \rangle$. App a_1 may be able to poison object o_2 even though it does not possess permission p_4 to do so; instead, it would use object o_1, app a_2, and app a_2's permission p_4. App a_1 cannot poison object o_3, since no path exists between them.

Consider the example cross-app information flow graph in Figure 2. Continuing the example from Section 1, suppose that app a_1 is a host-tracking app that has been compromised by an adversary; o_1 is the host store; a_2 is a routing app that has not been compromised; and o_2 is the flow entry store. The adversary does not have the ability to directly modify object o_2, because the app does not have permission to do so; if it did, an edge would exist from a_1 to o_2. However, the adversary can poison object o_1, since it is allowed to do so (*i.e.,* by permission p_1). Later, the routing app a_2, which has permissions that the adversary seeks (*i.e.,* any edge into o_2), reads from o_1 and uses information from o_1 to write to o_2.

4.4 Cross-App Poisoning Gadgets

Our methodology in Sections 4.1–4.3 conservatively captures how apps could influence data flowing through the shared control plane state, subject to a specified RBAC policy. Put simply, *what are the apps allowed to influence if they can read and write to such shared state?* However, such influences, represented as cross-app attack vectors, may not always exist in practice, since an app's source of data from the shared control plane state may not always causally influence what the app later writes to the control plane.

To account for that, we use static analysis techniques to identify relevant data flows present in apps that read from a permissioned data source and write to a permissioned data sink. We call such data flows *cross-app poisoning gadgets*, as one or more gadgets can be used to build sophisticated CAP attacks. CAP gadgets require a *triggering app* to start the chain reaction. We explain our specific methodology and implement proof-of-concept attacks for the Security-Mode ONOS SDN controller in Section 5.

5 CROSS-APP POISONING CASE STUDY: SECURITY-MODE ONOS

To show how prevalent CAP attacks are in practice, we study the Security-Mode ONOS SDN controller [61, 89]. We chose the ONOS framework because it is a representative example of a popular, production-quality controller used in industry by telecommunication service providers [62], among others. The ONOS framework is Java-based with publicly available source code[4] bundled with

[4]Throughout the paper, we use the ONOS v1.10.0 source code available at [60].

open-sourced apps. Security-Mode ONOS is a variant of the ONOS SDN controller with additional support for RBAC.

5.1 CAP Model for Security-Mode ONOS

5.1.1 Apps. The v1.10.0 release includes 64 bundled reference apps [59] as part of the ONOS codebase. Each app is an OSGi bundle that can be loaded into or removed from the controller at runtime as an internal app. Example apps include a reactive forwarding app (fwd), a routing app (routing), and a DHCP server (dhcp).

5.1.2 Permissions. By default, ONOS runs without any RBAC policies or enforcement; this makes execution of CAP attacks trivial, because nothing prevents an app from influencing any object in the shared control plane state. Instead, for the remainder of this paper, we evaluate Security-Mode ONOS, because it allows app developers to specify which permissions their apps need, and security practitioners can write RBAC policies that specify which roles apps have and what permissions each role has. Security-Mode ONOS includes 56 permissions named with `*_READ`, `*_WRITE`, and `*_EVENT` suffixes. We incorporate `*_READ` permissions into P_R and `*_WRITE` permissions into P_W. `*_EVENT` permissions register and de-register apps from event handlers, so we treat these permissions as equivalent to both read and write permissions.

5.1.3 Objects. ONOS follows the pattern of providing a "service class" (*e.g.,* FlowRuleService) that serves as an API for apps. Each service class has a respective "manager class" (*e.g.,* FlowRuleManager) that implements the service class. When the manager class is instantiated, it instantiates a respective "store class" (*e.g.,* FlowRuleStore) that stores the actual shared control plane state. That state is composed of "data class" instantiations (*e.g.,* objects of the classes FlowRule and FlowEntry). Each store is protected by limiting access via the manager class's methods (*e.g.,* getFlowEntries()), and, when apps call such methods, Security-Mode ONOS performs permission checks (*e.g.,* "Does the app have the `FLOWRULE_READ` permission according to the RBAC policy?"). ONOS also includes manager classes for the southbound API (*e.g.,* OpenFlowPacketContext).

We let each manager class represent an object in our model, given that a manager class encapsulates the methods and stores that represent access to and storage of the shared control plane state, respectively. As Security-Mode ONOS specifies permissions at the method level of granularity rather than at the "data class" level of granularity, we map these methods back to the manager classes when building the RBAC policy in the next section. For instance, an app that calls the getFlowEntries() method would need the `FLOWRULE_READ` permission, so our model would show an edge labeled with that permission from the FlowRuleManager object to the app in the cross-app information flow graph.

5.1.4 RBAC Policy. We assume that a practitioner sets up an RBAC policy of least privilege such that each app has the minimum set of permissions needed in order to carry out its functionality correctly. The 64 apps included with ONOS do not list the permissions that they would need if they were run with Security-Mode ONOS. We wrote a script that statically analyzed the ONOS codebase to find in which methods Security-Mode ONOS checked permissions. From there, we analyzed which apps used those methods in order to map the permissions that each app would need.

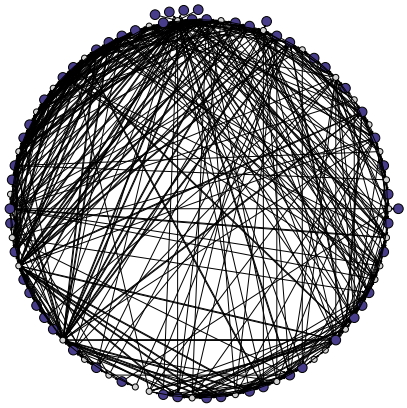


Figure 3: Cross-App Information Flow Graph $\mathcal{G}_{ONOS}$ using the 64 apps included with ONOS. Large points represent apps; small points represent objects in the shared SDN control plane state; and arrows represent permissions for apps to read from or write to objects.

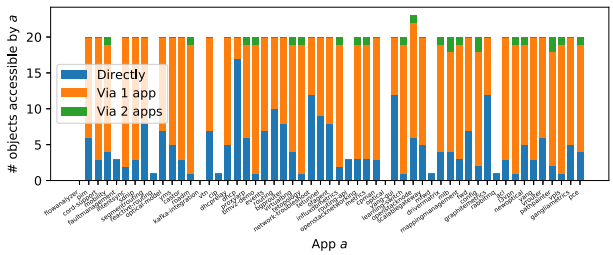


Figure 4: App to object accessibility (via shortest paths) in $\mathcal{G}_{ONOS}$ with 63 apps. Paths begin at a given app a.

Our result is a security reference policy for ONOS apps that enforces least privilege using RBAC and is called $\mathcal{R}_{ONOS}$. We found that Security-Mode ONOS permissions were enforced on 212 methods protected across 39 manager classes through the use of 38 of the available 56 permissions. Each manager class may implement more than one service class, so we included 67 service classes. (See Table 3 in Appendix A.2 for additional details.)

5.1.5 Cross-App Information Flow Graph. Using the security reference policy, we applied Algorithm 1 to generate the cross-app information flow graph $\mathcal{G}_{ONOS}$ for ONOS with all apps included.[5] Figure 3 shows the complete $\mathcal{G}_{ONOS}$ with 88 nodes[6] and 564 edges. To understand the connectivity of $\mathcal{G}_{ONOS}$, we looked at how many objects each app could directly and indirectly access (Figure 4) and how many apps each object could be accessed by, either directly or indirectly (Figure 5).[7] For both analyses, we removed an app named test from consideration, since it is used for testing ONOS functionality.

[5] We imagine that a practitioner would only load some subset of apps into the controller, so apps that have not been loaded should be removed from $\mathcal{G}_{ONOS}$ for analysis.
[6] Manager classes whose methods were not called by any app were not included in the cross-app information flow graph; thus, $|A| + |O| \neq 88$.
[7] A shortest path in $\mathcal{G}$ of length 3, for instance, corresponds to indirect accessibility via 1 app in Figure 4 or 1 object in Figure 5.

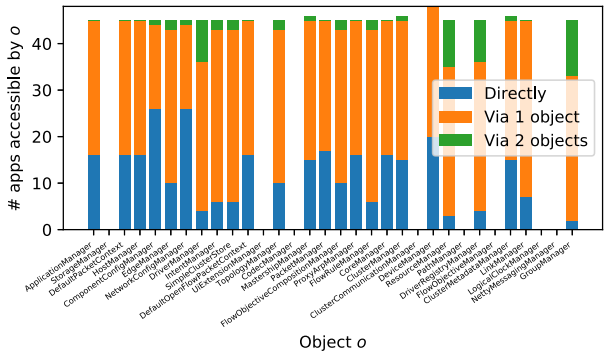


Figure 5: Object to app accessibility (via shortest paths) in $\mathcal{G}_{ONOS}$ with 63 apps. Paths begin at a given object o.

5.2 CAP Gadgets in Security-Mode ONOS

We further refine the results from Figures 4 and 5 by identifying a set of CAP gadgets in ONOS apps. Fortunately, all of the apps bundled with the ONOS codebase have publicly available source code that can be analyzed; while this is not strictly required to identify CAP gadgets, it simplifies the process. We used static analysis techniques to identify data flows that can be used to build CAP gadgets to instigate CAP attacks.

5.2.1 Methodology. We used JavaParser [82] to build an abstract syntax tree (AST) representation of each of the 63 ONOS apps, excluding the test app. Using the ASTs as inputs, we wrote a script to determine data flows within apps' methods from "sources" to "sinks" of interest through field-sensitive interprocedural data flow analysis. Such data flows represent an app's use of one control plane object to generate another control plane object. We defined *sources* as API read calls to permission-protected methods (*i.e.,* requiring a permission in P_R), and *sinks* as API write calls to permission-protected methods (*i.e.,* requiring a permission in P_W). We used P_R, P_W, and the list of 212 permission-protected methods found from our earlier analysis. We mapped the permission-protected methods to their respective permissions so that each source or sink is represented by a permission.

Although we used Java-specific tools to generate ASTs for ONOS apps, other tools such as CAST [76] for C/C++ or ast [25] for Python exist for controllers and apps in other languages.

5.2.2 Results. Table 1 shows the resulting cross-app poisoning gadgets, represented as (*source*, *app*, *sink*) tuples. One can chain gadgets together to form complex cross-app information flows. At a minimum, only one gadget is needed; any app that can write to a single gadget's source can launch a CAP attack. We summarize the behavioral takeaways and their consequences below:

(1) Five gadgets use the `APP_READ` source permission. In inspecting the apps' code, we found that the apps use the CoreService's methods to look up the mapping between the app's name (*e.g.,* `org.onosproject.fwd`) and a unique app ID (*e.g.,* `id=70`), and that the apps then subsequently use this app ID to take other control plane actions (*e.g.,* deleting all flow rules with the app ID `id=70`). If such assumptions about the trustworthiness of the app name and ID mapping are broken, faulty or malicious apps can cause systemic damage through CAP attacks *even if they have no permission to take such actions themselves.*

(2) Five gadgets use the `FLOWRULE_WRITE` sink permission. This would be expected, since most flow rule operations in ONOS are event-driven based on actions in the NB and SB APIs.

(3) Some objects are not affected by CAP attacks. We expect objects that are not related to maintaining network state (*e.g.,* objects for gathering statistics) to be unaffected.

5.3 Example Attack: Packet Modification and Flow Rule Insertion for Data Plane DoS

We now consider a proof-of-concept CAP attack that leverages the reactive forwarding app fwd to insert corrupted flow rules. We performed the attack using Security-Mode ONOS enabled with ONOS v1.10.0. (See Appendix A.1 for configuration details.)

5.3.1 Approach. We wrote a triggering app (trigger) to poison the view of the reactive forwarding app (fwd) so as to cause data plane denial-of-serivce (DoS). Our approach is similar to the attacks proposed by Dhawan *et al.* [15] and Lee *et al.* [39] to poison the view of the network, though we assume that malicious apps, rather than malicious switches or end hosts, cause the poisoning. Our triggering app minimally requires `PACKET_*` permissions and does not require `FLOWRULE_*` permissions. (See Appendix B for additional details.) The attack works as follows:

(1) The triggering app, to register itself with ONOS to receive incoming packets, uses its `PACKET_EVENT` permission. Upon receiving particular ARP requests, the app changes the ARP and Ethernet source addresses to an attacker's address.

(2) The forwarding app also registers for incoming packets. The forwarding app reads the packet by using the `PACKET_READ` permission to decide whether to generate flow rules.

(3) The forwarding app inserts the flow rule into the control plane using its `FLOWRULE_WRITE` permission. As a result, the flow rule becomes associated with the forwarding app because of fwd's `appId`.

5.3.2 Results. The flow rule based on corrupted information causes a data plane DoS attack from the victim's perspective. Because the forwarding app inserted the flow rule, ONOS identifies fwd as being responsible for the corresponding flow rule in its flow rule database. Thus, a practitioner investigating the DoS outage may incorrectly assign full blame to fwd, particularly since trigger is not assumed to have the ability to insert flow rules.

5.4 Remarks

We were able to systematically detect CAP gadgets (as described in Section 5.2) because the apps' source code was available, but this detection may not be an option with closed-source "black box" apps. Thus, practitioners need further insight into how apps behave in practice once they are activated within the SDN controller.

It is much easier to bypass RBAC permissions when apps are reading from or writing to many of the same shared SDN control plane state's objects. What is needed is a way to track information flow to capture how data are used *after* RBAC authorization is granted. By making access control decisions based not only on

Table 1: Static Analysis Results of CAP Gadgets for Security-Mode ONOS Apps.

Source ($p \in P_R$)	App ($a \in A$)	Sink ($p \in P_W$)	Attacker's capabilities if source data have been compromised by attacker
APP_READ	openstacknetworking	FLOWRULE_WRITE	Attacker modifies the app ID to remove all flows with a given app ID
APP_READ	openstacknode	CLUSTER_WRITE	Attacker modifies the app ID to make an app run for leader election in a different ONOS topic (*i.e.,* an app using ONOS's distributed primitives)
APP_READ	openstacknode	GROUP_WRITE	Attacker modifies the app ID to associate an app with a particular group handler
APP_READ	routing	CONFIG_WRITE	Attacker modifies the app ID to misapply a BGP configuration
APP_READ	sdnip	CONFIG_WRITE	Attacker modifies the app ID to misapply an SDN-IP encapsulation configuration
DEVICE_READ	newoptical	RESOURCE_WRITE	Attacker misallocates bandwidth resources based on a connectivity ID
DEVICE_READ	vtn	DRIVER_WRITE	Attacker misconfigures driver setup for a device (*i.e.,* switch)
DEVICE_READ	vtn	FLOWRULE_WRITE	Attacker misconfigures flow rules based on a device ID
HOST_READ	vtn	FLOWRULE_WRITE	Attacker misconfigures flow rules based on a host with a particular MAC address
PACKET_READ	fwd	FLOWRULE_WRITE	Attacker injects or modifies an incoming packet to poison a flow rule
PACKET_READ	learning-switch	FLOWRULE_WRITE	Attacker injects or modifies an incoming packet to poison a flow rule

the accessing app's role but also on the history of how data were generated, a practitioner can limit the extent to which apps are able to influence other apps while still maintaining the flexibility afforded by a shared state design.

6 INFORMATION FLOW CONTROL POLICIES

We consider information flow control (IFC) policies as they relate to detecting and preventing CAP attacks. We use a "floating label" approach based on Myers and Liskov's decentralized IFC model [53] and on previous IFC policies that use data provenance [66, 77]. In our policy model, a practitioner labels apps with *integrity tags*, resulting in each app's having its own *integrity label* composed of a subset of integrity tags. We assume that apps' label assignments cannot be modified by any actions that the apps take themselves, but that they can be changed out-of-band by practitioners as needed. Our IFC policy model for shared SDN control plane state integrity, denoted by $\mathcal{I} = (A, T, L, Ch, Re)$, consists of:

- A set of apps[8], denoted by $A = \{a_1, a_2, \ldots, a_x\}$.
- A set of integrity tags, denoted by $T = \{\tau_1, \tau_2, \ldots, \tau_t\}$.
- Integrity labels that map apps to a subset of integrity tags, denoted by $L : A \rightarrow \mathcal{P}(T)$, where $\mathcal{P}(T)$ is the power set of T.
- An enforcement check policy on when to check for violations, denoted by $Ch \in \{\texttt{READS}, \texttt{WRITES}\}$.
- A response to perform when information flow is violated, denoted by $Re \in \{\texttt{BLOCK}, \texttt{WARN}, \texttt{NONE}\}$.

An app's integrity label that is a superset relative to another app's integrity label has *higher integrity*; that is, if $L(a_i) \supseteq L(a_j)$, then a_i has integrity at least as high as that of a_j for $a_i, a_j \in A$ and $L(a_i), L(a_j) \in \mathcal{P}(T)$. We define an object's *integrity level*, denoted by $I(o)$ for $o \in O$, as the intersection of all integrity labels of apps that have helped generate that object. Formally, $I(o) = \bigcap_{i}^{n} L(a_i)$ for some set of apps $A_N = \{a_1, a_2, \ldots, a_n\}$ used in producing o. This means that the object's integrity level is as high as that of the lowest-integrity app that helped generate it.

7 ProvSDN

We now present our defense, ProvSDN. ProvSDN hooks all of the controller's API interfaces to collect provenance from apps, builds a provenance graph, and serves as an online reference monitor by checking API requests against the IFC policy $\mathcal{I}$. This allows us to prevent both known and unknown CAP attacks based on policy.

7.1 Data Provenance Model

Data provenance refers to the process of tracing and recording the origins of data and their movement. Provenance has been used to understand the flow of data in databases [2, 13, 22, 28, 86], operating systems [8, 45, 52, 67], mobile phones [3, 16], and browsers [40, 44]. Provenance can be used not just for IFC but also for information tracing, accountability, transparency, and compliance [51, 79].

We use the W3C PROV data model [47, 51], which defines provenance as a directed acyclic graph (DAG) that encodes the relationships between three elements (*i.e.,* vertices): *entities* are data objects processed by a system, *activities* are dynamic actions in the system, and *agents* are the principals that control system actions. Relations (*i.e.,* edges) describe the interactions between system elements. Entitities are *used* or *generated by* activities; activities are *associated with* agents; and activities may be *informed by* other activities. An advantage of storing provenance graphically is that it allows for efficient relational querying [7, 8, 27]. (See Table 4 in Appendix C for a visual representation of provenance objects and relations.)

7.1.1 Entities. We define *entities* as the objects from Section 6, which include the control plane's shared data structures that are being processed or generated by the SDN apps and controller. For ONOS, we define entities at the "data class" granularity as described in Section 5, since that definition captures fine-grained information about switches, hosts, and the network topology as well as flow rules, packets being processed, and OpenFlow messages sent or received. ProvSDN can also flexibly specify additional metadata to collect (*e.g.,* traffic match fields for a flow entry), as needed.

7.1.2 Activities. We define *activities* as the API calls and callbacks between SDN apps and the controller. For instance, these calls enable apps to process flow rules and OpenFlow messages.

7.1.3 Agents. We define *agents* as the principal identities of the apps, the switches, and the controller[9]. We treat switches as principal identities because, like apps that interact with the controller via

[8]For reasons explained in Section 7.1, we count switches as "apps."

[9]Internal controller services can interact with the shared SDN control plane state through event updates. We represent each internal controller service with its own agent; each of those agents performs operations on behalf of the controller agent.

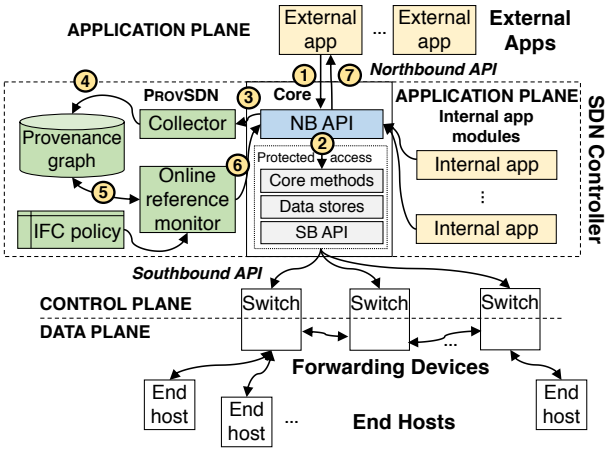


Figure 6: ProvSDN architecture showing an app calling the NB API. 1: An app makes a NB API request. 2: The NB API tentatively retrieves or inserts data related to the request. 3: The collector processes the call information. 4: The collector writes the provenance data to the provenance graph. 5: The online reference monitor checks the provenance graph for violations according to the IFC policy. 6: The IFC policy's response is returned to the NB API. 7: Depending on the response, the data may be returned to the app or may be written to the shared SDN control plane state .

the NB API, switches interact with the controller via the SB API. We attribute all activities (*i.e.,* API calls) to the agents that requested them, effectively identifying all activities of apps and switches that interact with the shared SDN control plane state.

7.2 System Components

Figure 6 shows the ProvSDN architecture. We assume that the provenance components are trusted and adequately secured.

7.2.1 Provenance collector. The provenance collector captures the API call information, such as which method was called, who called it, what data were used, and what data were subsequently generated. The collector also identifies relations and the agents, activities, and entities involved. From there, the collector converts the data into a W3C PROV-compliant graph. ProvSDN also collects information from SB API calls, given that some NB API calls cause packets in the data plane to be sent to the controller. ProvSDN hooks the SB API functions responsible for sending flow rules and processing incoming packets. That allows for association of incoming OpenFlow packets with the flow rules that caused them to be sent to the controller and ensures that the provenance graph correctly represents that association.

7.2.2 Online reference monitor. The online reference monitor checks the current provenance graph in real time against the IFC policy $\mathcal{I}$. For instance, suppose that the enforcement check policy Ch is `READS`. First, when data cross the API boundary for read requests, we consider that to be the equivalent to an attempt by a requesting app a_r to read object o. Next, we determine A_N by checking for the existence of paths from o to $\forall a \in A$. We check the policy $\mathcal{I}$ against 1) the label of the requesting app $L(a_r)$ and 2) the labels of the apps that the object previously encountered, or $\bigcap_i^n L(a_i)$. Finally, we apply the response Re, which can block the read request, warn the practitioner that the read request occurred, or do nothing. If a policy is violated and the response Re in the policy $\mathcal{I}$ is `BLOCK`, the relationship is removed[10] and the action is disallowed. Otherwise, the relationship is permanently added to the provenance graph.

7.2.3 Provenance graph. ProvSDN's provenance graph database enables online policy checking via the reference monitor, as well as offline investigation of previous events for network forensics.

7.3 Implementation

We implemented ProvSDN with ONOS v1.10.0. We describe our implementation details below.

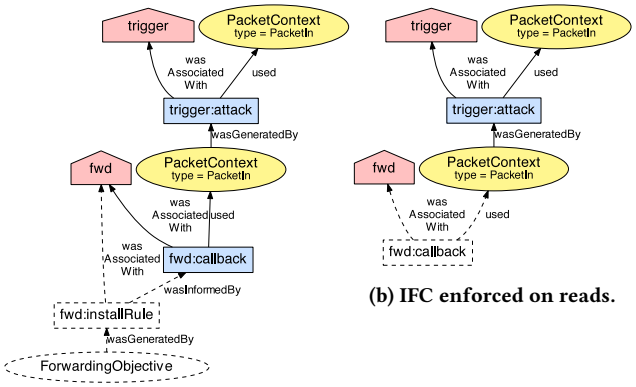
7.3.1 NB API. We found that the ONOS NB API was not well-defined and thus was subject to questions about whether apps could bypass provenance collection. To fix that, we used Doxygen [83] to identify all publicly accessible classes in ONOS by counting the number of references in the codebase to each of these classes; any class referenced by more than three other classes was deemed to be part of the NB API and properly exposed to SDN apps. Our static analysis identified 63 classes with 721 methods that we used as ONOS's NB API (*e.g.,* switch, host, link, and flow rule management). It also identified 194 classes with 1,405 methods that are internal to ONOS and should *not* be part of the NB API (*e.g.,* distributed storage primitives, and raw OpenFlow message handlers).

To prevent apps from bypassing provenance collection, we enforce internal method checking (step 2 of Figure 6). If an internal method call originates in another internal method, it is allowed; if it originates in an app, it is blocked. This forces apps to use the NB API through methods that capture provenance.

7.3.2 Provenance capture. The choice of programming language is important to ensure that access to controller internals is possible only through instrumented API calls. (See Appendix D for the challenges of implementing provenance on other controllers.) We found Java to work well in this regard by enforcing `private` or `public` access modifiers. By default, Java's controls are insufficient, because it is possible to override the declared access modifiers by using the `Reflection` API. Fortunately, static analysis can detect reflection use if apps are checked prior to being loaded.

7.3.3 Processing and storage. We implemented the ProvSDN provenance collector and online reference monitor in approximately 1,350 lines of Java code. We embedded approximately 420 provenance hooks throughout the ONOS codebase to call ProvSDN's provenance collector. Upon initialization, the collector imports the IFC policy $\mathcal{I}$ that the online reference monitor references when new provenance relations have been added. We stored provenance data in an internal JGraphT [56] graph structure for optimized graph search (*i.e.,* path existence) performance.

[10]To maintain an audit record, the relationship can remain in the provenance graph but be marked as not existing for the purpose of online graph queries.



(a) IFC enforced on writes.

(b) IFC enforced on reads.

Figure 7: Provenance graphs generated from example CAP attack described in Section 5. Dashed nodes and edges represent attempted actions blocked (but recorded) by ProvSDN.

7.4 Attack Evaluation

We evaluated ProvSDN's IFC capabilities using the attack described in Section 5.3. We prevent information flow from the triggering app (trigger) to the reactive forwarding app (fwd) by assigning different integrity tags to the apps. We set our IFC policy $\mathcal{I}$ as $T = \{\tau_1, \tau_2\}$, $L(\text{trigger}) = \{\tau_1\}$, $L(\text{fwd}) = \{\tau_1, \tau_2\}$, and $Re = \texttt{BLOCK}$. Since $L(\text{fwd}) \supset L(\text{trigger})$, fwd has higher integrity than trigger and is prevented from reading data generated by trigger. Packets sent from trigger and read by fwd, represented as PacketContext entities, have integrity levels $I(\text{packet}) = \{\tau_1\}$; ProvSDN computes $I(\text{packet})$ by checking path connectivity between entities and agents.

Figure 7 shows parts of the provenance graphs generated from the information flow attempts. If $Ch = \texttt{WRITES}$, IFC is enforced during write attempts, resulting in the process shown in Figure 7a. Similarly, if $Ch = \texttt{READS}$, IFC is enforced during read attempts, resulting in the process shown in Figure 7b. In both scenarios, the desired goal of the attacker (*i.e.,* to insert a corrupted flow rule) is blocked, albeit at different stages of the processing pipeline (depending on practitioner preference).

Suppose that the attack described in Section 5.3 had not been blocked and was allowed to occur. If the log files were verbose enough, a practitioner analyzing them might eventually be able to reconstruct the events that occurred. However, ProvSDN's provenance collection would make the investigation simpler even if IFC policies were not initially enforced. The practitioner issues a query to ProvSDN requesting information about the ForwardingObjective flow rule entity and receives the relevant ancestry (shown in Figure 7a). The graphical representation lets the practitioner start at the ForwardingObjective entity and trace back what data were used in generating the flow rule to see that trigger modified the original PacketContext entity. To prevent future occurrences, the practitioner installs the IFC policy described earlier.

7.5 Performance Evaluation

We evaluated ProvSDN's performance in an emulated environment running Open vSwitch 2.7.0 [24] software switches, which are commonly found in virtualized environments. We generated data plane packets so that they would be handled by the controller; this made ProvSDN collect, record, and query provenance. All experiments were performed on a four-core Intel Xeon E7-4850 2.0 GHz CPU with 16 GB of RAM running Ubuntu 16.04.2 LTS.

Table 2: ProvSDN Micro-Benchmark Latencies.

Operation	Average time per operation	Number of operations	Percent of total time
Collect	155.66 μs	23 067	1.38%
Write	11.15 μs	57 948	0.25%
IFC check	98.50 μs	544	0.02%
Internal check	44.67 μs	5 692 315	98.34%

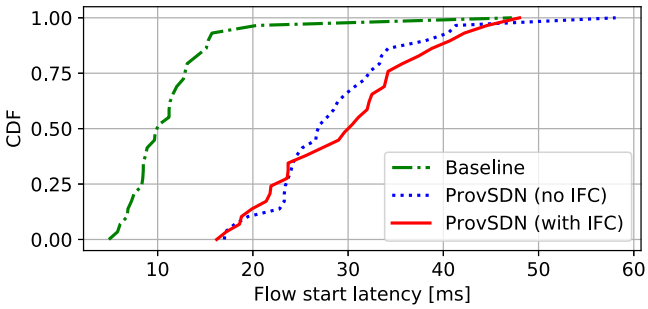


Figure 8: Flow start latency macrobenchmarks.

7.5.1 Macro-benchmarking. Our SDN macro performance metric of interest is *flow start latency*, which measures the time necessary for a data plane packet that does not match existing flow rules to be handled by the controller and apps. It represents the delays experienced from the end host perspective in reactive-based SDN configurations. The controller's packet handling will trigger several provenance events and checks (*e.g.,* new host event, topology change event, or flow insertion event).

Figure 8 shows the resulting latencies for a baseline without ProvSDN, for ProvSDN when IFC is not enforced, and for ProvSDN with IFC enforced. 30 trials were run for each of the three scenarios. The average latencies were 11.66 ms, 28.51 ms, and 29.53 ms, respectively. Although ProvSDN increases the baseline latency for packet handling, as more apps and internal controller services register to receive events, we note that the higher first-packet latency is amortized over longer flows, because subsequent packets matched to flow rules in switches do not need to go to the controller or to apps (or, by extension, to ProvSDN) for processing. Thus, ProvSDN needs to operate only on the relatively infrequent control plane state changes rather than on each individual packet of a flow.

7.5.2 Micro-benchmarking. We measured the additional latency overheads imposed by 1) collection of provenance, 2) writing of provenance to the provenance graph, and 3) performance of IFC checks by querying of the provenance graph. In addition, we measured 4) the latency imposed by enforcing the rule that apps cannot call internal controller methods (*i.e.,* the latency imposed by checking protected access as shown in step 2 of Figure 6). From Table 2, we see that internal method-checking operations impose most of

the additional latency (about 98% of total operations), even though they impose only a small additional latency per operation (44.67 μs on average). IFC checking is slower but infrequent, because the queries, in effect, test path connectivity between a source node (*i.e.,* an entity) and destination nodes (*i.e.,* the system's agents) in the provenance graph.

8 DISCUSSION

Extent to Which Controllers Are Affected by CAP. OpenDaylight [64] provides RBAC services based on the Apache Shiro Java Security Framework's permissions system, though RBAC services are not enabled by default. The current authorization scheme can be configured only after the controller starts and is "aimed towards supporting coarse-grained security policies" [70].

Floodlight [20] does not support RBAC and would thus be susceptible to CAP attacks. Floodlight provides core controller services similar to those of ONOS, such as LinkDiscoveryManager, TopologyService, and MemoryStorageSource. The MemoryStorageSource data store documentation notes that "all data is shared and there is no enforcement," [21] which would make CAP attacks trivial. SE-Floodlight [68] enforces RBAC but only on permissions for low-level switch operations rather than for app interactions such as those with which Security-Mode ONOS provides for ONOS.

Ryu [71], written in Python, does not support RBAC and would thus be trivially susceptible to CAP attacks. Python does not enforce public and private access protections.

Finer-Grained RBAC as CAP Mitigation. One way to reduce the control plane's attack surface is by implementing finer-grained RBAC. SDNShield [85], for instance, includes sub-method permissions such as allowing or denying flow entries based on IP source and destination prefixes. (See Appendix A for further details on how Security-Mode ONOS implements fine-grained permissions.) We can represent the finer-grained partitioning of permissions by considering finer-grained objects o in our cross-app information flow graph $\mathcal{G}$ and finer-grained permissions P in our RBAC model $\mathcal{R}$. Since the source code for SDNShield was not publicly available, we were not able to evaluate the extent to which finer-grained RBAC could help mitigate CAP attacks by using SDNShield. However, we surmise that finer-grained RBAC will still not solve problems such as reliance of system-wide apps (*e.g.,* a firewall app that protects an entire network) on trustworthy information about many objects.

Android. We compare the SDN network OS architecture with the Android mobile OS architecture, as both architectures include extensible third-party app ecosystems. While Android apps are sandboxed and communicate with each other through inter-process communication (IPC), SDN apps read from and write to a common shared control plane state over which access control (in practice) has been coarsely defined. The situation for SDN is more challenging than that of Android because Android apps can operate relatively independently of each other, but SDN architectures require greater coordination among SDN apps to ultimately maintain one main shared resource (*i.e.,* the data plane) through a limited number of data structures. This required coordination limits the effectiveness and practicality of sandboxing and IPC for SDN. As a result of the SDN shared-state design, maliciously generated data from one SDN app have significant repercussions for any other app that subsequently uses the data, or for the data plane.

Other IFC Mechanisms. Stack-based access control (SBAC) [5] and history-based access control (HBAC) [4] propose IFC for Java-based systems. Jif [54] is a Java extension for enforcing language-level IFC policies, but it has certain drawbacks. It would require retrofitting of all apps with IFC policy intents, would require app developers to know how to program IFC policies, and would not provide a record of information flow for later analysis. Dynamic taint analysis tracks information from "sources" entering the system to "sinks" leaving the system, but dynamic taint analysis is not as conducive to IFC because there may be a delay between the occurrence and the detection of an IFC violation [66]. We opted for data provenance techniques because provenance provides a historical record of information flow, its collection can be checked in real time, and its collection is agnostic to the controller's language.

For Android, TaintDroid [18] labels data from privacy-sensitive sources (*e.g.,* GPS, camera, or microphone) and applies labels as sensitive data propagate through program variables, files, and inter-process messages. However, TaintDroid does not capture the provenance of such interactions, and that limits further analysis. IPC Inspection [19], like ProvSDN, uses a low-watermark floating label policy [10] for Android to prevent permission re-delegation. Quire [16] tracks Android's IPC calls by annotating each call with apps that have processed the call. Quire is like ProvSDN in that one of its goals is to prevent confused deputy attacks, but since SDN architectures do not use IPC to exchange information, ProvSDN requires tracking and enforcement at the NB and SB API boundaries instead. Weir [55] enforces decentralized IFC for Android through polyinstantiation of applications and their components to reconcile different security contexts and to avoid label explosion. However, it is not clear whether such an approach would work with the limited data structures of the SDN shared state design.

For Web browsers, Bauer *et al.* [9] implemented and formally verified an IFC extension to the Chromium Web browser that uses light-weight taint tracking to track coarse-grained confidentiality and integrity labels across DOM elements and browser events. ProvSDN focuses on integrity-based attacks and collects full provenance metadata to reconstruct previous control plane states.

Limitations. ProvSDN's floating-label-based IFC design cannot prevent availability-based attacks in which low-integrity apps attempt to write to many objects to poison them so they cannot be read by high-integrity apps. The "self-revocation problem" in low-watermark systems [19, 26] demotes an agent's integrity level if the agent observes low-integrity data and then cannot modify data that it originally generated. The problem is partially mitigated in ProvSDN through fixed integrity labels for agents (*i.e.,* apps) and through implicit label propagation (*i.e.,* floating labels) for data objects. If availability-based attacks are of interest, ProvSDN can still be useful in identifying such behavior even without initially enforcing IFC, since ProvSDN will record such object poisoning. The provenance graph can be used to better inform practitioners in making decisions on whether such apps' behaviors are desirable and whether low-integrity apps should be removed.

ProvSDN with Security-Mode ONOS does not enforce separation of memory space since ONOS's OSGi-based container approach

does not enforce this separation. We rely upon Java's access modifiers to prevent apps from accessing private data structures. One alternative design approach would be to transparently separate each app into its own process and bridge API calls to the controller to enforce isolation by means of the underlying operating system, but this would require a significant redesign of the ONOS architecture. For language-based limitations, see Appendix D.

9 RELATED WORK

SDN Controller Security. Wen *et al.* [85] note four classes of SDN controller attacks: data plane intrusions, information leakage, rule manipulation, and apps' attacking of other apps. The authors propose SDNShield for fine-grained RBAC and app isolation policies to prevent inter-app attacks, but as shown in the cross-app information flow graph for Security-Mode ONOS in Figure 3, an app sandboxing policy is too restrictive in practice, because apps necessarily rely on information generated by other apps in order to function correctly. The authors claim that the logs from SDNShield can be used for offline forensic analysis, but it is unclear whether such logs explicitly show information flow and, if so, how they do. With ProvSDN, we allow practitioners to flexibly specify their intents about each app's integrity assumptions to enforce a desired IFC policy in real time, and our provenance-based approach captures a history of information flow by design.

Security-Mode ONOS [89] extends the ONOS controller to include API method-level RBAC enforcement. Rosemary [75] isolates applications by running each application as an individual process. SE-Floodlight [68] hardens the control plane by enforcing hierarchical RBAC policies and logging events through an auditing subsystem. These systems neither explicitly the track information flow necessary for detecting CAP attacks nor enforce IFC policies in real time as can be done with ProvSDN. FRESCO [74] allows for enforcement of hierarchical flow-rule deconfliction to ensure that non-security applications cannot undo actions taken by security applications; however, this is limited to the controller–switch interface and provides no protection from CAP attacks.

An orthogonal approach would be to use secure-by-construction controllers that utilize languages whose type systems guarantee properties such as app composability [1, 23, 49, 50, 84]. In such systems, the controller acts more as a language runtime than as an operating system, and applications are written in a formal language and composed using logical operators. We consider such controllers to be sufficiently different from operating-system-like controllers that they are out of the scope of this paper.

SDN App Security. Malicious apps are arguably one of the most severe threats to SDN security, as the dynamic configurations available in SDN architectures can make it challenging to determine whether the network's state is (or was) correct according to policy [14]. Several efforts [39, 42] have outlined attacks similar to CAP attacks that affect Floodlight, ONOS, and OpenDaylight, though they did not consider the case in which apps that do not have permission to take actions co-opt other apps that do have such permissions. The authors of [39, 42] propose to use permission checking, static analysis, and dynamic analysis as defenses; ProvSDN goes beyond that approach by enforcing IFC policies. Other SDN attacks, particularly those that rely upon data plane information to make control plane decisions, exist in the literature and are too numerous to list here; we refer the reader to Lee *et al.* [41].

Network Verification and Testing. An approach complementary to that of ProvSDN would be to test whether, and/or formally verify that, controller or application behavior falls within a set of invariants. VeriFlow [36] and NetPlumber [35], like ProvSDN, perform real-time invariant checks, but they implicitly assume a monolithic controller and do not capture the history of information flow that ProvSDN does. NICE [11] verifies that an application cannot install flow rules that violate a set of constraints, but does not consider controller–application interactions. DELTA [41], ATTAIN [81], and BEADS [34] provide SDN testing frameworks but are necessarily incomplete because of their reliance on fuzzing.

Provenance in SDN. Provenance-based approaches are just beginning to emerge in the SDN context. GitFlow [17] tracks network state by committing state changes with a version control system, but it requires extensive retrofitting of all apps and data plane elements, does not operate in real time, and does not account for malicious apps. Ujcich *et al.* [80] consider how provenance can be used to detect faults from benign application interactions in an offline manner, but do not consider malicious applications or online attack detection. Wu *et al.* [87] leverage meta-provenance to facilitate automated repair of a network. Bates *et al.* [6] demonstrate a way to improve a previous approach [90] by using SDN to enforce the monitoring of host-to-host communication. However, those three efforts considered communications only in the data plane rather than the control plane.

Provenance tracing is of demonstrated value to network forensic efforts. Zhou *et al.* [90] consider the task of identifying malicious nodes in a distributed system. Chen *et al.* [12] diagnose network problems by reasoning about the differences between two provenance graphs, while in other work the *absence* of provenance relationships has been used to explain network behaviors [88].

10 CONCLUSION

We have demonstrated CAP attacks that allow SDN apps to poison the integrity of the network view seen by the SDN controller and other SDN apps. CAP attacks take advantage of the lack of IFC protections within SDN controllers. We show how RBAC solutions to date are inadequate for solving this problem. Using the Security-Mode ONOS controller as a case study, we also demonstrate ProvSDN, a provenance-based defense that captures control plane information flow and enforces online IFC policies for SDN apps that access or modify the SDN control plane.

ACKNOWLEDGMENTS

The authors thank our shepherd Adwait Nadkarni and the anonymous reviewers for their helpful comments, which improved this paper; Ahmed Fawaz, the PERFORM research group, and the STS research group 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 Maryland Procurement Office under Contract No. H98230-18-D-0007 and by the National Science Foundation under Grant Nos. CNS-1657534 and CNS-1750024.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 113–126. https://doi.org/10.1145/2578855.2535862

[2] B Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. 2014. A Generic Provenance Middleware for Database Queries, Updates, and Transactions. In *Proceedings of the Theory and Practice of Provenance (TaPP '14)*.

[3] Michael Backes, Sven Bugiel, and Sebastian Gerling. 2014. Scippa: System-centric IPC Provenance on Android. In *Proceedings of ACSAC '14*. ACM, 36–45. https://doi.org/10.1145/2664243.2664264

[4] Anindya Banerjee and David A. Naumann. 2005. History-based Access Control and Secure Information Flow. In *Proceedings of CASSIS '04*. Springer-Verlag, Berlin, Heidelberg, 27–48. https://doi.org/10.1007/978-3-540-30569-9_2

[5] Anindya Banerjee and David A. Naumann. 2005. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming* 15, 2 (March 2005), 131–177. https://doi.org/10.1017/S0956796804005453

[6] Adam Bates, Kevin Butler, Andreas Haeberlen, Micah Sherr, and Wenchao Zhou. 2014. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. In *Proceedings of USENIX SENT '14*. USENIX Association.

[7] Adam Bates, Devin J. Pohly, and Kevin R. B. Butler. 2016. Secure and Trustworthy Provenance Collection for Digital Forensics. In *Digital Fingerprinting*, Cliff Wang, Ryan M. Gerdes, Yong Guan, and Sneha Kumar Kasera (Eds.). Springer New York, 141–176. https://doi.org/10.1007/978-1-4939-6601-1_8

[8] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-system Provenance for the Linux Kernel. In *Proceedings of USENIX Security Symposium '15*. USENIX Association, 319–334.

[9] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of NDSS '15*. Internet Society. https://doi.org/10.14722/ndss.2015.23295

[10] K. J. Biba. 1975. *Integrity Considerations for Secure Computer Systems*. Technical Report MTR-3153. MITRE Corporation. www.dtic.mil/dtic/tr/fulltext/u2/a039324.pdf

[11] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Proceedings of NSDI '12*, Vol. 12. 127–140. http://dl.acm.org/citation.cfm?id=2228298.2228312

[12] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of ACM SIGCOMM '16*. ACM, 115–128. https://doi.org/10.1145/2934872.2934910

[13] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. 2005. DBNotes: A Post-it System for Relational Databases Based on Provenance. In *Proceedings of the 2005 ACM Special Interest Group on Management of Data Conference (SIGMOD'05)*.

[14] M. C. Dacier, H. König, R. Cwalinski, F. Kargl, and S. Dietrich. 2017. Security Challenges and Opportunities of Software-Defined Networking. *IEEE Security & Privacy* 15, 2 (March 2017), 96–100. https://doi.org/10.1109/MSP.2017.46

[15] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Proceedings of NDSS '15*. Internet Society. https://doi.org/10.14722/ndss.2015.23064

[16] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of USENIX Security '11*. USENIX Association, 23–23.

[17] Abhishek Dwaraki, Srini Seetharaman, Sriram Natarajan, and Tilman Wolf. 2015. GitFlow: Flow Revision Management for Software-defined Networks. In *Proceedings of ACM SOSR '15*. ACM, Article 6, 6 pages. https://doi.org/10.1145/2774993.2775064

[18] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[19] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Conference on Security Symposium (SEC '11)*. USENIX Association, Berkeley, CA, USA, 22–22. http://dl.acm.org/citation.cfm?id=2028067.2028089

[20] Floodlight. 2018. (2018). http://www.projectfloodlight.org/

[21] Project Floodlight. 2018. Floodlight Controller: MemoryStorageSource (Dev). (2018). https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343633/MemoryStorageSource+Dev

[22] Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. 2002. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *Proceedings of SSDBM '02*. https://doi.org/10.1109/SSDM.2002.1029704

[23] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. 2010. Frenetic: A High-level Language for OpenFlow Networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO '10)*. ACM, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/1921151.1921160

[24] Linux Foundation. 2018. Open vSwitch. (2018). https://www.openvswitch.org/

[25] Python Software Foundation. 2018. ast—Abstract Syntax Trees. (2018). https://docs.python.org/3/library/ast.html

[26] Timothy Fraser. 2000. LOMAC: Low Water-Mark integrity protection for COTS environments. In *Proceeding of the 2000 IEEE Symposium on Security and Privacy (S&P '00)*. 230–245. https://doi.org/10.1109/SECPRI.2000.848460

[27] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of Middleware '12*. Springer–Verlag New York, 101–120. https://doi.org/10.1007/978-3-642-35170-9_6

[28] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model Through Query Rewriting. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE '09)*. https://doi.org/10.1109/ICDE.2009.15

[29] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. 2008. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review* 38, 3 (2008), 105–110. https://doi.org/10.1145/1384609.1384625

[30] Norm Hardy. 1988. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review* 22, 4 (Oct. 1988), 36–38. http://dl.acm.org/citation.cfm?id=54289.871709

[31] Hewlett-Packard Enterprise. 2018. HPE SDN app store. https://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps. (Aug. 2018).

[32] Hewlett-Packard Enterprise. 2018. Software Defined Networking. (Aug. 2018). https://www.hpe.com/us/en/networking/sdn.html

[33] International Data Corporation. 2016. SDN Market to Experience Strong Growth Over Next Several Years, According to IDC. https://www.idc.com/getdoc.jsp?containerId=prUS41005016. (3 Feb. 2016).

[34] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. 2017. BEADS: Automated Attack Discovery in OpenFlow-based SDN Systems. In *Proceedings of RAID '17*. https://doi.org/10.1007/978-3-319-66332-6_14

[35] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of NSDI '13*. 99–111. http://dl.acm.org/citation.cfm?id=2482626.2482638

[36] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. VeriFlow: Verifying Network-wide Invariants in Real Time. *ACM SIGCOMM Computer Communication Review* 42, 4 (Sept. 2012), 467–472. https://doi.org/10.1145/2377677.2377766

[37] Diego Kreutz, Fernando M.V. Ramos, and Paulo Veríssimo. 2013. Towards Secure and Dependable Software-defined Networks. In *Proceedings of ACM HotSDN '13*. 55–60. https://doi.org/10.1145/2491185.2491199

[38] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (Jan 2015), 14–76. https://doi.org/10.1109/JPROC.2014.2371999

[39] Chanhee Lee and Seungwon Shin. 2016. SHIELD: An Automated Framework for Static Analysis of SDN Applications. In *Proceedings of ACM SDN-NFV Security '16*. ACM, 29–34. https://doi.org/10.1145/2876019.2876026

[40] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of NDSS '13*. Internet Society.

[41] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. 2017. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proceedings of NDSS '17*. Internet Society. https://doi.org/10.14722/ndss.2017.23457

[42] Seungsoo Lee, Changhoon Yoon, and Seungwon Shin. 2016. The Smaller, the Shrewder: A Simple Malicious Application Can Kill an Entire SDN Environment. In *Proceedings of ACM SDN-NFV Security '16*. ACM, 23–28. https://doi.org/10.1145/2876019.2876024

[43] R. J. Lipton and L. Snyder. 1977. A Linear Time Algorithm for Deciding Subject Security. *J. ACM* 24, 3 (July 1977), 455–464. https://doi.org/10.1145/322017.322025

[44] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th USENIX Security Symposium*.

[45] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of NDSS '16*.

[46] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[47] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV Family of Specifications for Modelling Provenance Metadata. In *Proceedings of ACM EDBT '13*. 773–776. https://doi.org/10.1145/2452376.2452478

[48] Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. 2013. Corybantic: Towards the Modular Composition of SDN Control Programs. In *Proceedings of ACM HotNets '13*. ACM, Article 1, 7 pages. https://doi.org/10.1145/2535771.2535795
[49] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 217–230. https://doi.org/10.1145/2103621.2103685
[50] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. 2013. Composing Software Defined Networks. In *Proceedings of NSDI '13*, Vol. 13. 1–13. http://dl.acm.org/citation.cfm?id=2482626.2482629
[51] Luc Moreau and Paul Groth. 2013. *Provenance: an introduction to PROV*. Synthesis Lectures on the Semantic Web: Theory and Technology, Vol. 3. Morgan & Claypool Publishers. 1–129 pages. https://doi.org/10.2200/S00528ED1V01Y201308WBE007
[52] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware Storage Systems. In *Proceedings of USENIX ATC '06*. USENIX Association, 1.
[53] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of ACM SOSP '97*. ACM, 129–142. https://doi.org/10.1145/268998.266669
[54] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2018. Jif: Java + Information Flow. (2018). http://www.cs.cornell.edu/jif/
[55] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical DIFC Enforcement on Android. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC '16)*. USENIX Association, Berkeley, CA, USA, 1119–1136. http://dl.acm.org/citation.cfm?id=3241094.3241181
[56] Barak Naveh. 2018. JGraphT. (2018). https://jgrapht.org/
[57] Open Networking Foundation. 2018. Creating Security-Mode Compatible ONOS Applications. (2018). https://wiki.onosproject.org/display/ONOS/Creating+Security-Mode+compatible+ONOS+applications
[58] Open Networking Foundation. 2018. Enabling Security-Mode ONOS. (2018). https://wiki.onosproject.org/display/ONOS/Enabling+Security-Mode+ONOS
[59] Open Networking Foundation. 2018. GitHub – onos/apps at 1.10.0. (2018). https://github.com/opennetworkinglab/onos/tree/1.10.0/apps
[60] Open Networking Foundation. 2018. GitHub – opennetworkinglab/onos at 1.10.0. (2018). https://github.com/opennetworkinglab/onos/tree/1.10.0
[61] Open Networking Foundation. 2018. ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out. (Aug. 2018). https://onosproject.org/
[62] Open Networking Foundation. 2018. ONOS In Action. (2018). https://onosproject.org/in-action/
[63] Open Networking Foundation. 2018. Security Advisories: ONOS. (2018). https://wiki.onosproject.org/display/ONOS/Security+advisories
[64] OpenDaylight. 2018. Home - OpenDaylight. (Aug. 2018). https://www.opendaylight.org/
[65] Hitesh Padekar, Younghee Park, Hongxin Hu, and Sang-Yoon Chang. 2016. Enabling Dynamic Access Control for Controller Applications in Software-Defined Networks. In *Proceedings of SACMAT '16*. ACM, 51–61. https://doi.org/10.1145/2914642.2914647
[66] T. Pasquier, J. Singh, D. Eyers, and J. Bacon. 2017. CamFlow: Managed Data-sharing for Cloud Services. *IEEE Transactions on Cloud Computing* PP, 99 (2017), 1–1. https://doi.org/10.1109/TCC.2015.2489211
[67] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-fidelity Whole-system Provenance. In *Proceedings of ACM ACSAC '12*. ACM, 259–268. https://doi.org/10.1145/2420950.2420989
[68] Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. 2015. Securing the Software-Defined Network Control Layer. In *Proceedings of NDSS '15*. Internet Society. https://doi.org/10.14722/ndss.2015.23222
[69] POX. 2018. POX. (Aug. 2018). https://github.com/noxrepo/pox
[70] OpenDaylight Project. 2018. OpenDaylight Documentation: Authentication, Authorization and Accounting (AAA) Services. (2018). https://docs.opendaylight.org/en/stable-boron/user-guide/authentication-and-authorization-services.html
[71] Ryu SDN Framework Community. 2018. Ryu SDN Framework. (Aug. 2018). http://osrg.github.io/ryu/
[72] A. Sabelfeld and A. C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121
[73] S. Scott-Hayward, S. Natarajan, and S. Sezer. 2016. A Survey of Security in Software Defined Networks. *IEEE Communications Surveys Tutorials* 18, 1 (2016), 623–654. https://doi.org/10.1109/COMST.2015.2453114
[74] SW Shin, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular composable security services for software-defined networks. In *Proceedings of NDSS '13*. Internet Society.
[75] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. 2014. Rosemary: A Robust, Secure, and High-performance Network Operating System. In *Proceedings of ACM CCS '14*. ACM, 78–89. https://doi.org/10.1145/2660267.2660353
[76] The Flick Team. 2018. C Abstract Syntax Tree (CAST) Representation. (2018). http://www.cs.utah.edu/flux/flick/current/doc/guts/gutsch6.html
[77] Dave (Jing) Tian, Adam Bates, Kevin R.B. Butler, and Raju Rangaswami. 2016. ProvUSB: Block-level Provenance-Based Data Protection for USB Storage Devices. In *Proceedings of ACM CCS '16*. ACM, 242–253. https://doi.org/10.1145/2976749.2978398
[78] Y. Tseng, M. Pattaranantakul, R. He, Z. Zhang, and F. NaÃŕt-Abdesselam. 2017. Controller DAC: Securing SDN controller with dynamic access control. In *Proceedings of IEEE ICC '17*. 1–6. https://doi.org/10.1109/ICC.2017.7997249
[79] Benjamin E. Ujcich, Adam Bates, and William H. Sanders. 2018. A Provenance Model for the European Union General Data Protection Regulation. In *Provenance and Annotation of Data and Processes*, Khalid Belhajjame, Ashish Gehani, and Pinar Alper (Eds.). Springer International Publishing. https://doi.org/10.1007/978-3-319-98379-0
[80] Benjamin E. Ujcich, Andrew Miller, Adam Bates, and William H. Sanders. 2017. Towards an accountable software-defined networking architecture. In *Proceedings of IEEE NetSoft '17*. IEEE. https://doi.org/10.1109/NETSOFT.2017.8004206
[81] Benjamin E. Ujcich, Uttam Thakore, and William H. Sanders. 2017. ATTAIN: An Attack Injection Framework for Software-Defined Networking. In *Proceedings of IEEE/IFIP DSN '17*. IEEE. https://doi.org/10.1109/DSN.2017.59
[82] Danny van Bruggen. 2018. JavaParser: For Parsing Java Code. (2018). http://javaparser.org/
[83] Dimitri van Heesch. 2018. Doxygen: Generate documentation from source code. (2018). http://www.stack.nl/~dimitri/doxygen/
[84] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. 2012. Procera: a language for high-level reactive network control. In *Proceedings of ACM HotSDN '12*. ACM, 43–48. https://doi.org/10.1145/2342441.2342451
[85] Xitao Wen, Bo Yang, Yan Chen, Chengchen Hu, Yi Wang, Bin Liu, and Xiaolin Chen. 2016. SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets. In *Proceedings of IEEE/IFIP DSN '16*. IEEE, 121–132. https://doi.org/10.1109/DSN.2016.20
[86] Jennifer Widom. 2004. *Trio: A System for Integrated Management of Data, Accuracy, and Lineage*. Technical Report 2004-40. Stanford InfoLab. http://ilpubs.stanford.edu:8090/658/
[87] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2015. Automated Network Repair with Meta Provenance. In *Proceedings of ACM HotNets '15*. ACM, Article 26, 7 pages. https://doi.org/10.1145/2834050.2834112
[88] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2014. Diagnosing Missing Events in Distributed Systems with Negative Provenance. In *Proceedings of ACM SIGCOMM '14*. ACM, 383–394. https://doi.org/10.1145/2619239.2626335
[89] Changhoon Yoon, Seungwon Shin, Phillip Porras, Vinod Yegneswaran, Heedo Kang, Martin Fong, Brian O'Connor, and Thomas Vachuska. 2017. A Security-Mode for Carrier-Grade SDN Controllers. In *Proceedings of ACM ACSAC '17*. ACM, 461–473. https://doi.org/10.1145/3134600.3134603
[90] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure Network Provenance. In *Proceedings of ACM SOSP '11*. ACM, 295–310. https://doi.org/10.1145/2043556.2043584

A SECURITY-MODE ONOS DETAILS

Security-Mode ONOS specifies permissions at the 1) bundle, 2) application, 3) API, and 4) network (*i.e.,* header space) levels [89]. We considered the API level permissions in our RBAC analysis in Section 5.1, since it was an appropriate level of granularity for discussing the shared SDN control plane data structures' permissions. Although the Security-Mode ONOS paper describes network-level permissions that would allow for finer granularities beyond API level permissions (*e.g.,* `FLOWRULE_READ` with packets matching an IP source address within `10.0.0.0/24`), we were not able to find the relevant code in the ONOS repository [60] that implemented such permissions at the time of writing.

A.1 Configuration

Security-Mode ONOS requires the installation of the Apache Felix Framework security extensions and a reconfiguration of Apache Karaf prior to running the controller [58]. It is expected that app developers must create a manifest of necessary permissions for an

app in order for it to be allowed to be used when running Security-Mode ONOS [89]. Such a manifest subsequently is included with the app and is verified when the app is installed [57].

In addition to our static analysis script (see Section 5.1.4) that we used to determine which permissions apps would need to run with Security-Mode ONOS, we encountered other permissions that needed to be set at the bundle and application levels. In particular, the interactions with the OSGi framework required that we allow the `org.osgi.framework.ServicePermission` and the `org.osgi.framework.AdminPermission` permissions for all OSGi bundles so that the apps could interact with the core ONOS services; not doing so produced silent failures.

A.2 App, Permission, and Object Details

Table 3 enumerates the specific apps, read permissions, write permissions, and objects that we used in our CAP model for Security-Mode ONOS.

B SELECTED CODE FOR REACTIVE FORWARDING APP

Figure 9 shows the relevant Java code portions for the reactive forwarding app fwd. The reactive forwarding app requires `PACKET_*` permissions to set up a packet processor (Line 5, Figure 9) and to process such packets (Line 9, Figure 9), in addition to the `FLOWRULE_WRITE` permission to emit flow rules into the data plane (Line 17, Figure 9). We also permitted fwd to have the `APP_*` (Line 4, Figure 9), `CONFIG_*`, `DEVICE_READ`, `TOPOLOGY_READ`, `INTENT_*`, and `HOST_READ` permissions to ensure fwd's proper operation.

Note that any flows generated from fwd are attributed to fwd through the `fromApp(appId)` method (Line 16, Figure 9), in spite of the fact that fwd's decisions may be based on data generated by other apps. In the case of the attack from Section 5.3, trigger poisons such data before they arrive to fwd (Line 9, Figure 9).

C W3C PROV-DM REPRESENTATIONS

Table 4 summarizes the visual representations of the W3C PROV data model's provenance objects and relations [51]. The basic PROV object classes are `Agent`, `Activity`, and `Entity`. The basic PROV relation classes that we use for ProvSDN are `wasGeneratedBy`, `wasAttributedTo`, `used`, `wasInformedBy`, `wasAssociatedWith`, and `actedOnBehalfOf`.

D IMPLEMENTING ProvSDN ON OTHER CONTROLLERS

Provenance is effective only if an adversary cannot bypass the collection system. We note that the feasibility of satisfying this requirement depends significantly on the language used to implement the SDN controller. Certain language features may aid (*e.g.,* private/public declarators) or hinder (*e.g.,* lack of memory safety) the ability to instrument all communication paths between apps and the controller. Here, we discuss what challenges exist if ProvSDN were to be implemented on other SDN controllers.

```
1  public class ReactiveForwarding {
2     public void activate(...) {
3        ...
4        appId = coreService.registerApplication("org.onosproject.
             ↪ fwd");
5        packetService.addProcessor(processor, PacketProcessor.
             ↪ director(2));
6        ...
7     }
8     private class ReactivePacketProcessor implements
          ↪ PacketProcessor {
9        public void process(PacketContext context) {
10          ...
11          installRule(context,...);
12       }
13    }
14    private void installRule(PacketContext context,...) {
15       ...
16       ForwardingObjective forwardingObjective =
             ↪ DefaultForwardingObjective.builder().withSelector(
             ↪ selectorBuilder.build()).withTreatment(treatment).
             ↪ withPriority(flowPriority).withFlag(
             ↪ ForwardingObjective.Flag.VERSATILE).fromApp(appId).
             ↪ makeTemporary(flowTimeout).add();
17       flowObjectiveService.forward(context.inPacket().
             ↪ receivedFrom().deviceId(), forwardingObjective)
18    }
19 }
```

Figure 9: Selected reactive forwarding app code during app activation and during packet processing for inserting flow rules. Lines with permissioned calls are highlighted in gray.

D.1 Java-Based Open-Source Controllers

In addition to ONOS, Floodlight [20], SE-Floodlight [68], and OpenDaylight [64] are all implemented in Java. Classes in Java can have member variables be declared as `private` or `protected`, which prevents other, potentially malicious classes from directly manipulating such variables. All interactions must be through `public` method invocations that can be instrumented to collect provenance data. In addition, Java is memory-safe, barring the exploitation of vulnerabilities against the JVM itself. This ensures that an attacker cannot, for instance, corrupt a reference to point to a sensitive object's `private` or `protected` member variables.

As noted earlier, Java's `Reflection` API should be disabled to prevent overriding the declared access modifiers. Furthermore, the bytecode of compiled Java classes can be modified at class-load time, and several libraries are available to facilitate this process. This may allow an attacker to remove provenance collection code, or induce other unwanted behaviors into other classes. In order to collect complete provenance information, both reflection and byte code rewriting should be disabled. For example, static analysis can detect use of such methods and refuse to load classes which exploit these features.

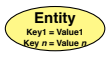
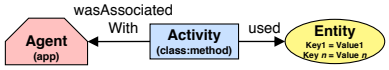
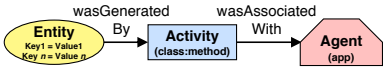
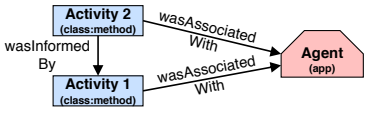
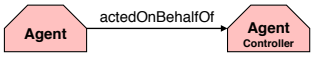
D.2 Python-Based Open-Source Controllers

Several SDN controllers, including Ryu [71] and POX [69], are written in Python. Python does not enforce private data structures that are only accessible to their containing class. All objects can directly manipulate the attributes of all other objects and do not need to go through getter and setter calls that could otherwise enforce instrumentation. As such, it difficult to support internal

Table 3: Partial RBAC Model for Security-Mode ONOS and Included ONOS Apps.

Apps: A = {acl, actn-mdsc, bgprouter, bmv2-demo, castor, cip, config, cord-support, cpman, dhcp, dhcprelay, drivermatrix, events, faultmanagement, flowanalyzer, flowspec-api, fwd, gangliametrics, graphitemetrics, influxdbmetrics, intentsync, iptopology-api, kafka-integration, l3vpn, learning-switch, mappingmanagement, metrics, mfwd, mlb, mobility, netconf, network-troubleshoot, newoptical, ofagent, openroadm, openstacknetworking, openstacknode, optical, optical-model, pathpainter, pce, pcep-api, pim, proxyarp, rabbitmq, reactive-routing, restconf, roadm, routing, routing-api, scalablegateway, sdnip, segmentrouting, tenbi, test, tetopology, tetunnel, virtualbng, vpls, vrouter, vtn, yang, yang-gui, yms}
Read permissions: P_R = {APP_READ, APP_EVENT, CONFIG_READ, CONFIG_EVENT, CLUSTER_READ, CLUSTER_EVENT, CODEC_READ, DEVICE_KEY_EVENT, DEVICE_KEY_READ,DEVICE_READ, DEVICE_EVENT, DRIVER_READ, EVENT_READ, FLOWRULE_READ, FLOWRULE_EVENT, GROUP_READ, GROUP_EVENT, HOST_READ, HOST_EVENT, INTENT_READ, INTENT_EVENT, LINK_READ, LINK_EVENT, PACKET_READ, PACKET_EVENT, PARTITION_READ, PARTITION_EVENT, RESOURCE_READ, RESOURCE_EVENT, REGION_READ, STATISTIC_READ, TOPOLOGY_READ, TOPOLOGY_EVENT, TUNNEL_READ, TUNNEL_EVENT, UI_READ}
Write permissions: P_W = {APP_EVENT, APP_WRITE, CONFIG_WRITE, CONFIG_EVENT, CLUSTER_WRITE, CLUSTER_EVENT, CODEC_WRITE, CLOCK_WRITE, DEVICE_KEY_EVENT, DEVICE_KEY_WRITE, DEVICE_EVENT, DRIVER_WRITE, EVENT_WRITE, FLOWRULE_WRITE, FLOWRULE_EVENT, GROUP_WRITE, GROUP_EVENT, HOST_WRITE, HOST_EVENT, INTENT_WRITE, INTENT_EVENT, LINK_WRITE, LINK_EVENT, MUTEX_WRITE, PACKET_WRITE, PACKET_EVENT, PERSISTENCE_WRITE, PARTITION_EVENT, RESOURCE_WRITE, RESOURCE_EVENT, STORAGE_WRITE, TOPOLOGY_EVENT, TUNNEL_WRITE, TUNNEL_EVENT, UI_WRITE}
Objects: O = {ApplicationManager, ClusterCommunicationManager, ClusterManager, ClusterMetadataManager, CodecManager, ComponentConfigManager, CoreEventDispatcher, CoreManager, DefaultOpenFlowPacketContext, DefaultPacketContext, DeviceKeyManager, DeviceManager, DriverManager, DriverRegistryManager, EdgeManager, FlowObjectiveCompositionManager, FlowObjectiveManager, FlowRuleManager, FlowStatisticManager, GroupManager, HostManager, IntentManager, LinkManager, LogicalClockManager, MastershipManager, NettyMessagingManager, NetworkConfigManager, PacketManager, PartitionManager, PathManager, PersistenceManager, ProxyArpManager, RegionManager, ResourceManager, SimpleClusterStore, StatisticManager, StorageManager, TopologyManager, UiExtensionManager}

Table 4: SDN Shared Control Plane State Semantics Using W3C PROV-DM.

Object or Event	W3C PROV-DM Representation
Control plane object with attributes	
App method or function call	
App, controller, or switch identity	
App reading object from the shared control plane	
App writing object to the shared control plane	
Intra-app method or callback method	
Internal service on behalf of controller	

apps while maintaining guarantees about complete provenance collection, outside of instrumenting the Python interpreter itself. One option is to move controller apps to discrete processes that communicate only over inter-process communication primitives. This would allow provenance collection at the cost of higher latency.

D.3 C/C++-Based Open-Source Controllers

Controllers written in C or C++, such as Rosemary [75] and NOX [29], support private data structures and allow provenance to be collected by instrumenting getters and setters. Unfortunately, neither language is memory-safe. This is a particularly severe problem for handling malicious apps. Not only could controller code contain exploitable bugs, but malicious apps themselves may deliberately include vulnerabilities that they exploit locally in order to gain arbitrary read/write access to memory. This clearly bypasses provenance collection and may even have more severe repercussions if the malicious app can, for example, make system calls.

D.4 Closed-Source Controllers

Collecting provenance data as discussed here implicitly requires the ability to instrument code, which is not possible for closed source controllers such as HP's VAN [32]. However, possible future work could leverage verbose log files to gain insight into interactions between the controller and apps.