

© 2020 Benjamin E. Ujcich

SECURING THE SOFTWARE-DEFINED NETWORKING CONTROL PLANE
BY USING CONTROL AND DATA DEPENDENCY TECHNIQUES

BY

BENJAMIN E. UJCICH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair
Assistant Professor Adam Bates
Professor Carl A. Gunter
Professor Ravishankar K. Iyer

ABSTRACT

Software-defined networking (SDN) fundamentally changes how network and security practitioners design, implement, and manage their networks. SDN decouples the decision-making about traffic forwarding (*i.e.*, the control plane) from the traffic being forwarded (*i.e.*, the data plane). SDN also allows for network applications, or apps, to programmatically control network forwarding behavior and policy through a logically centralized control plane orchestrated by a set of SDN controllers. As a result of logical centralization, SDN controllers act as network operating systems in the coordination of shared data plane resources and comprehensive security policy implementation.

SDN can support network security through the provision of security services and the assurances of policy enforcement. However, SDN's programmability means that a network's security considerations are different from those of traditional networks. For instance, an adversary who manipulates the programmable control plane can leverage significant control over the data plane's behavior.

In this dissertation, we demonstrate that the security posture of SDN can be enhanced using control and data dependency techniques that track information flow and enable understanding of application composability, control and data plane decoupling, and control plane insight. We support that statement through investigation of the various ways in which an attacker can use control flow and data flow dependencies to influence the SDN control plane under different threat models. We systematically explore and evaluate the SDN security posture through a combination of runtime, pre-runtime, and post-runtime contributions in both attack development and defense designs.

We begin with the development a conceptual accountability framework for SDN. We analyze the extent to which various entities within SDN are accountable to each other, what they are accountable for, mechanisms for assurance about accountability, standards by which accountability is judged, and the consequences of breaching accountability. We discover significant research gaps in SDN's accountability that impact SDN's security posture. In particular, the results of applying the accountability framework showed that more control plane attribution is necessary at different layers of abstraction, and that insight motivated the remaining work in this dissertation.

Next, we explore the influence of apps in the SDN control plane's secure operation. We find that existing access control protections that limit what apps can do, such as role-based access controls, prove to be insufficient for preventing malicious apps from damaging control plane operations. The

reason is SDN’s reliance on shared network state. We analyze SDN’s shared state model to discover that benign apps can be tricked into acting as “confused deputies”; malicious apps can poison the state used by benign apps, and that leads the benign apps to make decisions that negatively affect the network. That violates an implicit (but unenforced) integrity policy that governs the network’s security. Because of the strong interdependencies among apps that result from SDN’s shared state model, we show that apps can be easily co-opted as “gadgets,” and that allows an attacker who minimally controls one app to make changes to the network state beyond his or her originally granted permissions. We use a data provenance approach to track the lineage of the network state objects by assigning attribution to the set of processes and agents responsible for each control plane object. We design the PROVSDN tool to track API requests from apps as they access the shared network state’s objects, and to check requests against a predefined integrity policy to ensure that low-integrity apps cannot poison high-integrity apps. PROVSDN acts as both a reference monitor and an information flow control enforcement mechanism.

Motivated by the strong inter-app dependencies, we investigate whether implicit data plane dependencies affect the control plane’s secure operation too. We find that data plane hosts typically have an outsized effect on the generation of the network state in reactive-based control plane designs. We also find that SDN’s event-based design, and the apps that subscribe to events, can induce dependencies that originate in the data plane and that eventually change forwarding behaviors. That combination gives attackers that are residing on data plane hosts significant opportunities to influence control plane decisions without having to compromise the SDN controller or apps. We design the EVENTSCOPE tool to automatically identify where such vulnerabilities occur. EVENTSCOPE clusters apps’ event usage to decide in which cases unhandled events should be handled, statically analyzes controller and app code to understand how events affect control plane execution, and identifies valid control flow paths in which a data plane attacker can reach vulnerable code to cause unintended data plane changes. We use EVENTSCOPE to discover 14 new vulnerabilities, and we develop exploits that show how such vulnerabilities could allow an attacker to bypass an intended network (*i.e.*, data plane) access control policy. This research direction is critical for SDN security evaluation because such vulnerabilities could be induced by host-based malware campaigns.

Finally, although there are classes of vulnerabilities that can be removed prior to deployment, it is inevitable that other classes of attacks will occur that cannot be accounted for ahead of time. In those cases, a network or security practitioner would need to have the right amount of after-the-fact insight to diagnose the root causes of such attacks without being inundated with too much information. Challenges remain in 1) the modeling of apps and objects, which can lead to overestimation or underestimation of causal dependencies; and 2) the omission of a data plane model that causally links control and data plane activities. We design the PICO SDN tool to mitigate causal dependency modeling challenges, to account for a data plane model through the use of the data plane topology to

link activities in the provenance graph, and to account for network semantics to appropriately query and summarize the control plane's history. We show how prior work can hinder investigations and analysis in SDN-based attacks and demonstrate how PICO SDN can track SDN control plane attacks.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First, I would like to thank my Ph.D. co-advisors, Prof. William H. Sanders and Prof. Adam Bates, for their tireless support, guidance, and advice in helping to plan the research direction of this dissertation. Both gave me the freedom to pursue my research interests, the environment to become an independent researcher, and the space to discuss any questions about research, careers, and life. I am grateful to Prof. Ravishankar K. Iyer and Prof. Carl A. Gunter for their advice as Ph.D. committee members. I would also like to thank Prof. Andrew Miller, Prof. Klara Nahrstedt, Prof. Tarek F. Abdelzaher, Prof. Romit Roy Choudhury, Prof. Indranil Gupta, Prof. David M. Nicol, and Prof. Michael Bailey for their research discussions throughout my time at Illinois.

Much of this dissertation's work came about from three summer internships at MIT Lincoln Laboratory in Lexington, Massachusetts, which included external collaborators from MIT Lincoln Laboratory, Northeastern University, and Princeton University. I am indebted to Samuel Jero, Richard Skowrya, Steven R. Gomez, Hamed Okhravi, Anne Edmundson, Bryan C. Ward, Nathan H. Burow, Prof. Cristina Nita-Rotaru, James Landry, and Roger I. Khazan for their help, discussions, support, and friendship. Those significant and long-lasting collaborations throughout those summers and during the intervening academic years were instrumental in shaping this dissertation's research direction. I am forever thankful for my external colleagues who encouraged me in challenging times to persevere and to make this dissertation's research contributions the best that they could be.

I have been blessed to have had a wonderful group of friends and colleagues at Illinois with whom I have had many discussions about research and about life in general. In the Performability Engineering Research Group (PERFORM), I would like to thank Uttam Thakore, Mohammad Nouredine, Carmen Cheh, Ahmed Fawaz, Michael Rausch, Atul Bohara, Ronald Wright, Varun Badrinath Krishna, David Huang, Brett Feddersen, Ken Keefe and Gabriel Weaver. In the Secure & Transparent Systems Laboratory (STS), I would like to thank Wajih Ul Hassan, Pubali Datta, Riccardo Paccagnella, Dawei Wang, Noor Michael, Akul Goyal, Sneha Gaur, Saad Hussain, and Jai Pandey. I would additionally like to thank Qi Wang, Arjun Athreya, Subho Banerjee, Saurabh Jha, Yoga Varatharajah, Hui Lin, Homa Alemzadeh, Zane Ma, Joshua Reynolds, Deepak Kumar, Paul Murley, Seoung-kyun (Simon) Kim, Kartik Palani, and Elizabeth Reed.

Many of the research directions that I pursued during my time at Illinois were inspired by practi-

tioners who shared insights on various operational challenges. Debbie Fligor and Wayland Morgan in Technology Services at Illinois provided me with a better understanding of the networking, security, and privacy challenges of large enterprise environments. I would also like to thank David Grogan at the University of Illinois System's Ethics and Compliance Office for his discussions about the university system's approaches to data protection.

I am particularly grateful to Jenny Applequist, Jan Progen, and Jamie Hutchinson for their editorial assistance with this dissertation and related publications. I am appreciative of the wonderful support staff at Illinois, including Dawn Cheek-Wooten, Kelli Anderson, Linda Morris, Erica Kennedy, and Amy Irle, for their assistance with any question or need in the Coordinated Science Laboratory and Information Trust Institute. I am also indebted to Mandy Wisehart and Kathy Atchley (at Illinois), and Sue Haslett (at Carnegie Mellon University) for their incredible help with the preparation of my faculty and industry applications.

My research journey was inspired early on by several people during my time as an undergraduate and high school student. Prof. Kuang-Ching Wang and Prof. Harlan Russell at Clemson University encouraged me in my research pursuits and in the then-nascent SDN architecture. Dan Schmiedt and Brian Parker at Clemson University allowed me to observe the operational challenges of running a university network, which inspired my research interest in the operational challenges of network debugging, troubleshooting, and security. Several summer internships with great colleagues at the GENI Project Office at Raytheon BBN Technologies exposed me to various networking research activities being performed across the country. Prof. Varavut Limpasuvan at Coastal Carolina University inspired my interest as a high school student in the value and enjoyment of research.

Finally, I would like to thank my parents. I am grateful for their time spent listening to my challenges and successes throughout the dissertation process, and I am thankful for their unconditional and endless love.

This dissertation is based upon work supported by the Army Research Office under Award No. W911NF-13-1-0086; by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement number FA8750-11-2-0084; by the Department of Defense under Air Force Contract No. FA8721-05-C-0002; by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001; by the National Science Foundation under Grant Nos. CNS-1657534 and CNS-1750024; by the Maryland Procurement Office under Contract No. H98230-18-D-0007; by generous support from MIT Lincoln Laboratory; and by generous support from the Roy J. Carver Fellowship provided by the Roy J. Carver Charitable Trust. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the sponsoring organization.

TABLE OF CONTENTS

LIST OF TABLES	x
LIST OF FIGURES	xi
LIST OF ALGORITHMS	xii
LIST OF ABBREVIATIONS	xiii
CHAPTER 1 INTRODUCTION	1
1.1 Overview	1
1.2 Contributions	3
1.3 Organization	6
CHAPTER 2 BACKGROUND	7
2.1 SDN Architecture	7
2.2 SDN Architecture Attacks	13
2.3 SDN Architecture Defenses	19
CHAPTER 3 ACCOUNTABILITY IN SOFTWARE-DEFINED NETWORKING	27
3.1 Introduction	27
3.2 Designing an Accountable SDN Architecture	29
3.3 Case Study: Accountable SDN Applications	34
3.4 Conclusion	36
CHAPTER 4 CONTROL PLANE CROSS-APP POISONING	37
4.1 Introduction	37
4.2 Threat Model	40
4.3 Challenges	41
4.4 Cross-App Poisoning	42
4.5 Cross-App Poisoning Case Study: Security-Mode ONOS	46
4.6 Information Flow Control Policies	53
4.7 PROVSDN	54
4.8 Discussion	60
4.9 Related Work	63
4.10 Conclusion	64

CHAPTER 5	CONTROL PLANE EVENT-BASED VULNERABILITIES	65
5.1	Introduction	65
5.2	Challenges	69
5.3	EVENTSCOPE Overview	71
5.4	Event Use Analysis	73
5.5	Event Flow Analysis	79
5.6	Implementation	85
5.7	ONOS Vulnerability Evaluation Results	86
5.8	Discussion	94
5.9	Related Work	96
5.10	Conclusion	98
CHAPTER 6	CONTROL PLANE CAUSAL ANALYSIS	99
6.1	Introduction	99
6.2	Challenges	101
6.3	PicoSDN Provenance Model	106
6.4	PicoSDN Threat Model	112
6.5	PicoSDN Design	112
6.6	Implementation	120
6.7	Evaluation	121
6.8	Discussion	126
6.9	Related Work	127
6.10	Conclusion	128
CHAPTER 7	CONCLUSIONS	129
7.1	Review of Contributions	129
7.2	Overall Takeaways	130
7.3	Future Research	133
APPENDIX A	PUBLICATIONS RELATED TO THE DISSERTATION	135
APPENDIX B	ProvSDN	137
B.1	Security-Mode ONOS Details	137
B.2	Selected Code for Reactive Forwarding App	139
B.3	W3C PROV-DM Representations	139
B.4	Implementing ProvSDN on Other Controllers	139
APPENDIX C	EVENTSCOPE	143
C.1	ONOS Application Structure	143
C.2	ONOS Event Flow Graph Example	147
C.3	Number of Clusters and Detection Rate	147
REFERENCES		149

LIST OF TABLES

2.1	Known Attacks on the SDN Architecture.	14
2.2	Runtime Defenses for the SDN Architecture.	21
3.1	Summary of Design Considerations and Properties for SDN Accountability.	28
4.1	Static Analysis Results of CAP Gadgets for Security-Mode ONOS Apps.	51
4.2	PROVSDN Micro-Benchmark Latencies.	59
5.1	Event Listener Vulnerabilities Based on Event Flow Graph Analysis and Event Use Filtering ($\tau = 0.90$).	87
6.1	Nodes in the PICO SDN Provenance Graph Model.	107
6.2	Edges (Relations) in the PICO SDN Provenance Graph Model.	107
6.3	List of PICO SDN Hooks.	120
B.1	Partial RBAC Model for Security-Mode ONOS and Included ONOS Apps.	138
B.2	SDN Shared Control Plane State Semantics Using W ₃ C PROV-DM.	141

LIST OF FIGURES

2.1	SDN architecture overview and app interactions via service APIs and event callbacks.	8
2.2	SDN architecture with event listeners and event dispatchers.	11
4.1	Example of a cross-app information flow graph \mathcal{G}	45
4.2	Cross-app information flow graph \mathcal{G}_{ONOS}	48
4.3	App to object accessibility (via shortest paths) in \mathcal{G}_{ONOS}	49
4.4	Object to app accessibility (via shortest paths) in \mathcal{G}_{ONOS}	49
4.5	PROVSDN architecture showing an app calling the NB API.	55
4.6	Provenance graphs generated from example CAP attack.	58
4.7	Flow start latency macrobenchmarks.	59
5.1	Cross-plane attack example.	69
5.2	EVENTSCOPE architecture overview.	72
5.3	ONOS event use matrix.	77
5.4	Dendrogram representation of ONOS network event type similarity among apps. .	78
5.5	Event flow graph of fwd's packet processor.	82
5.6	Component analysis performance results.	85
5.7	Partial event flow graph showing vulnerable code paths used in CVE-2018-12691. .	89
5.8	Partial event flow graph showing vulnerable code paths used in CVE-2019-11189. . .	92
6.1	Topology of the CVE-2018-12691 attack scenario.	102
6.2	Data, process, and agency provenance of the CVE-2018-12691 attack scenario. . . .	103
6.3	Data partitioning models for flow rules.	109
6.4	Comparison of execution partitioning models.	110
6.5	Data plane model.	111
6.6	PicoSDN architecture overview with example workflow.	112
6.7	PicoSDN latency performance results.	122
6.8	Relevant features of the host migration attack's graph.	124
6.9	Relevant features of the graph from the cross-app attack.	126
B.1	Selected reactive forwarding app code.	140
C.1	Abbreviated code structure of an example ONOS network application.	144
C.2	Event flow graph of ONOS with core service components and several apps.	145
C.3	ONOS apps' candidate and valid vulnerabilities as a function of clustering threshold τ	147

LIST OF ALGORITHMS

4.1	Cross-App Information Flow Graph Generation	44
5.1	Candidate Vulnerability Generation	74
5.2	Event Flow Graph Generation	81
5.3	Vulnerability Validation	84
6.1	Data Plane Model	115
6.2	Common Ancestry Trace	116
6.3	Iterative Backward-Forward Trace	117
6.4	Network Activity Summarization	118
6.5	Network Identifier Evolution	119

LIST OF ABBREVIATIONS

API	Application programming interface
ARP	Address Resolution Protocol
AS	Autonomous system
AST	Abstract syntax tree
BDDP	Broadcast Domain Discovery Protocol
BGP	Border Gateway Protocol
CAP	Cross-app poisoning
CVE	Common Vulnerabilities and Exposures
DAG	Directed acyclic graph
DHCP	Dynamic Host Configuration Protocol
DoS	Denial of service
DPID	Datapath identifier
HMAC	Hash-based message authentication code
ICMP	Internet Control Message Protocol
IDS	Intrusion detection system
IFC	Information flow control
IP	Internet Protocol
IPC	Inter-process communication
IPv4	Internet Protocol version 4
LLDP	Link-layer Discovery Protocol
MAC	Media access control

MLS	Multi-level security
NB API	Northbound application programming interface
NOS	Network operating system
ODL	OpenDaylight
ONOS	Open Network Operating System
OSGi	Open Services Gateway initiative
OVS	Open vSwitch
PROV-DM	W ₃ C Provenance Data Model
RBAC	Role-based access control
REST	Representational state transfer
ROP	Return-oriented programming
SAT	Boolean satisfiability problem
SB API	Southbound application programming interface
SMT	Satisfiability modulo theories problem
SDN	Software-defined networking
STRIDE	Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege
TCAM	Ternary content-addressable memory
TCB	Trusted computing base
TCP	Transmission Control Protocol
TOCTTOU	Time-of-check-to-time-of-use
VLAN	Virtual local area network
W ₃ C	World Wide Web Consortium

CHAPTER 1

INTRODUCTION

1.1 Overview

Software-defined networking (SDN) is an emerging paradigm that provides flexible design of and control over computer networks. SDN has seen rapid commercialization to date, having been incorporated into a variety of application domains spanning industrial utilities, cloud computing, and telecommunication providers, among others [1]. As a result, it has been estimated that the market value of SDN will total more than \$12 billion by 2022 [2]. In contrast to traditional legacy networks, SDN decouples the decision-making of traffic forwarding (*i.e.*, the *control plane*) from the traffic being forwarded (*i.e.*, the *data plane*), logically centralizes the decision-making into an *SDN controller* composed of software processes, and exposes a set of network abstractions and API services to allow for development of *network applications* that extend the control plane’s functionality [1].

Given that the SDN architecture’s centralization allows a network operator to “see” all network activities [3], SDN has been proposed as a network security service that augments the functionality of existing host-based security services. For instance, SDN’s flexibility in defining network forwarding rules allows for the implementation of fine-grained network security policies [4, 5]. In contrast to separate middleboxes (*e.g.*, firewalls) that operate independently of the network’s forwarding devices (*e.g.*, switches and routers), the same network security functionality can be composed as network applications that react to network state changes and ensure that the proper network access control policies for hosts are enforced.

While SDN can implement network security services to further secure hosts on networks, the security of SDN, too, must be understood in order to systematically evaluate the overall SDN security posture relative to that of legacy networks. The SDN architecture introduces several new security considerations, such as attacks on controllers, the lack of trust between network applications and controllers, and the lack of trusted forensics tools for control plane insight [6]. Those issues, among others, have become more pronounced as simple SDN controllers have grown into complex and full-fledged programmable *network operating systems*.

Much like traditional desktop operating systems, network operating systems are responsible for the coordination and security of resources shared among applications and users. For networks,

the shared resources are the network’s control plane and the network’s various abstractions. SDN controllers operating as network operating systems, such as OpenDaylight (ODL) and the Open Network Operating System (ONOS), typically provide 1) a set of *network abstractions* for reasoning about network objects; 2) a set of *core services* and API calls that allow network applications to interact with a *shared SDN control plane state*, which comprises topological information and forwarding rule behavior; and 3) an *event-based* notification system that dispatches events to interested network applications and core services upon control plane state changes.

Given that the most popular SDN controllers have been designed as network operating systems, we consider the traditional challenges of secure operating system design as applied to the SDN context. That approach allows us to evaluate the security of SDN and the overall SDN security posture. We want to understand the unique characteristics of SDN that pose new challenges in the design of secure network operating systems, and we focus in particular on control plane insight and information flow within the control plane because of the control plane’s systemic effect on network operations. Taking SDN characteristics into account, we formulate three research questions about SDN control plane security vis-à-vis operating system design, as follows.

First, to what extent does the SDN design of composable network applications impact overall SDN security properties? We compare SDN to different domains for which operating systems have been designed and deployed. For instance, mobile operating systems, such as Android, can allow applications to operate relatively independently as sandboxes and to communicate through well-defined inter-process communication (IPC) [7]. In contrast, the SDN architecture inherently focuses on the network as a single shared resource,¹ and most security solutions to date have focused on authorization, permissions, and access control over that shared resource [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. In network operating systems, network applications share information through the use of API calls to core services, and these calls generate or use data that comprise the shared SDN control plane state. That inter-app communication mechanism via the shared state creates security challenges based on the potential for information flow from lower-integrity network applications to higher integrity network applications.

Second, to what extent does the SDN design of control and data plane decoupling impact overall SDN security properties? Compared to legacy networks, a touted benefit of SDN is its decoupling of the control and data planes into separate concerns and abstractions [1]. One consequence of that design decision is that the design allows for the centralization of decision-making within the controller and for the use of “dumb” forwarding devices (*i.e.*, switches and routers) to implement the forwarding. Although control planes can be proactively configured, reactive control planes make

¹Network “slicing” is possible through the use of network hypervisors, such as FlowVisor [8], that are positioned above an SDN controller or network operating system. However, network hypervisors suffer from scalability challenges [9] and have been largely supplanted by SDN controllers that account for multi-tenant operators and virtual networks.

decisions based on data plane information; for instance, control plane decisions can be made based on data from incoming packets that were generated by network hosts. Control plane decisions made from untrusted sources, such as spoofed packets, can have systemic effects in the control and data planes [22]. Thus, the apparent decoupling of planes belies the implicit information flow, which has far-reaching and unintended network security ramifications.

Third, what design considerations are necessary for greater control plane insight? For instance, to guarantee coverage over all control plane activities in order to take preventative actions against undesired behavior, mechanisms such as reference monitors [23] should ensure that interposition over calls and events to enforce policy is possible. However, current SDN controllers and network operating system designs do not provide a central location that can interpose over all such activities. Furthermore, given a shared-state design, the modeling of network abstractions and controller processes significantly affects the ability to reason about past activities for troubleshooting, root cause analysis, and attack understanding. However, current SDN controllers provide neither efficient mechanisms for accurate and precise tracking and understanding of causal dependencies, nor the precise agency representations necessary for attribution.

1.2 Contributions

It is our thesis that:

The security posture of a software-defined network can be enhanced using control and data dependency techniques that track information flow and enable understanding of application composability, control and data plane decoupling, and control plane insight.

This dissertation explores SDN control plane security with a focus on the design of secure network operating systems. Our goal is to study the security and accountability² of the control plane as a primary objective, and we use control and data dependency techniques to solve the associated research questions. In particular, we leverage *data provenance*, or the metadata about how data were used and generated, as a core methodology in our exploration. Data provenance is frequently represented graphically: nodes represent data objects, system activities, and system principals; and edges represent relations among such objects, activities, and principals. The resulting graphical structure is amenable to causal dependency reasoning. We also leverage *static analysis* to aid in the identification of security-relevant control and data flows within the control plane’s core services and network applications.

²*Accountability*, as used in this dissertation, refers to attribution mechanisms that enhance “non-repudiation, deterrence, fault isolation, intrusion detection and prevention, and after-action recovery and legal action” [24].

Throughout the dissertation, we use the ONOS SDN controller [25] as a running case study. ONOS is an open-source SDN controller developed in part by the Open Networking Foundation's Open Networking Lab and supported as a Linux Foundation project. The ONOS code base is freely available online [26], which has allowed us to examine and extend its source code. ONOS also incorporates an extensive API and event-based notification system among its core services' components and network applications. The ONOS code base also provides numerous network applications for analysis, and they serve as an underlying dataset for evaluating the efficacy of our various approaches.

Our specific contributions in this dissertation are the following:

- C1** We applied a conceptual framework of accountability to SDN to identify the agents, system entities, processes, and standards involved in network accountability assurance.
- C2** We illustrated the need for accountability through a practical case study scenario of SDN network applications.
- C3** We identified the information flow control (IFC) integrity problem in SDN, *i.e.*, cross-app poisoning (CAP). We demonstrated that malicious apps can utilize a lack of information flow protections to poison the control plane's state and escalate privilege.
- C4** We systematically identified CAP attack vulnerabilities, given a specified RBAC policy, by modeling the SDN control plane's allowed information flows.
- C5** We designed 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.
- C6** We implemented and evaluated CAP attacks and PROVSDN with the ONOS controller.
- C7** We designed an automated approach for analyzing SDN control plane event use by applications that identifies likely missing event handling and checks whether this lack of event handling, in combination with activities of other apps, can cause data-plane effects.
- C8** We created the event flow graph data structure, which allows for succinct identification of (a) event dispatching, event listening, and API use among SDN components, as well as (b) a context in which vulnerabilities can be realized.
- C9** We implemented our vulnerability discovery tool, EVENTSCOPE, in Java and Python.
- C10** We discovered and validated 14 new vulnerabilities in ONOS that escalate data plane access.
- C11** We designed an approach to the dependency explosion problem for SDN attack provenance that utilizes event listeners as units of execution.

C12 We designed an approach to the incomplete dependency problem for SDN attack provenance that incorporates a data plane model and tracking of network identifiers.

C13 We designed and implemented PICO SDN, which we use with ONOS to evaluate SDN attacks and to demonstrate PICO SDN’s causal analysis benefits.

C14 We evaluated the performance and security of PICO SDN using recent SDN attacks.

In combination, data provenance and static analysis provide insight into an SDN control plane’s control and data flows prior to, during, and after runtime to ensure security properties and to strengthen the overall SDN security posture. Thus, we consider our research contributions on data provenance and static analysis temporally:

- **Runtime:** We used data provenance to build a graphical model of the lineage of objects comprising the shared SDN control plane state as that state changes over time (**C5** and **C13**). We used a reference monitor that referenced the provenance graph to enforce control plane information flow control (IFC). The IFC was based on an integrity policy of interest that prevented high-integrity network applications from being corrupted by low-integrity network applications.
- **Pre-runtime:** We used static analysis to find “gadgets” of data flows that began at permissioned API read calls in network applications and flowed to permissioned API write calls (**C4**). Such “gadgets” can be used by low-integrity network applications to influence high-integrity network applications implicitly through the shared SDN control plane state. We also used static analysis techniques to study the extent to which information originated in the data plane (*e.g.*, incoming packets to be processed), traversed API calls and event dispatches within core services and network applications, and influenced control plane state changes that affected the data plane (*e.g.*, flow rule installation) (**C8**).
- **Post-runtime:** We used data provenance techniques to answer network-relevant questions for root cause analysis (**C14**). Key benefits of organizing past network state provenance in a graphical model include the ability to discard noncausal activities from past history and the ability to understand the effects of undesired activities on other activities, both of which simplify a practitioner’s task of root cause analysis after a security attack has occurred.

In addition to a temporal categorization, we can also consider our research contributions from the attacker’s and the defender’s perspectives:

- **Attack development:** We investigated a class of integrity-based information flow vulnerabilities among network applications, the exploitation of which we call *cross-app poisoning*,

that act as confused deputies (C3). We demonstrated that such attacks were possible through the use of a malicious app that instigated a series of cross-app poisoning “gadgets” (C4). We also investigated the class of information flow attacks that leverage data plane input to cause changes (via the control plane) to the data plane. We call such attacks *cross-plane event-based attacks* (C7). We found 14 new cross-plane event-based vulnerabilities in ONOS network applications, demonstrated several exploits that bypassed intended security policies, and registered 8 Common Vulnerabilities and Exposures (CVE) identifiers with MITRE (C10).

- **Defense design:** We designed provenance-based defense mechanisms and implemented them by extending the ONOS codebase and writing ONOS network applications to collect and query provenance data (C5 and C13). We hooked relevant API calls and event dispatches within core services so that we would be able to interpose on all relevant calls and events. We designed a provenance model that mitigates challenges in the modeling of the network’s causal dependencies (C11 and C12). We built additional data provenance analysis tools to provide after-the-fact root cause analysis insight into collected runtime provenance (C13).

1.3 Organization

The rest of the dissertation is organized as follows:

- Chapter 2 provides background on the SDN architecture’s components, the known security attacks against the architecture, and the defenses against such attacks.
- Chapter 3 proposes an accountability framework for the SDN architecture and points out gaps in the existing body of effort to secure the architecture.
- Chapter 4 presents the cross-app poisoning problem and the runtime defense tool PROVSDN for information flow control.
- Chapter 5 presents the cross-plane event vulnerability problem and the pre-runtime defense tool EVENTSCOPE for vulnerability discovery.
- Chapter 6 presents the challenges to control and data plane causal analysis and the post-runtime defense tool PICO SDN for causal analysis.
- Chapter 7 concludes the dissertation and provides overall takeaways and future research directions.

A list of publications related to this dissertation can be found in Appendix A.

CHAPTER 2

BACKGROUND

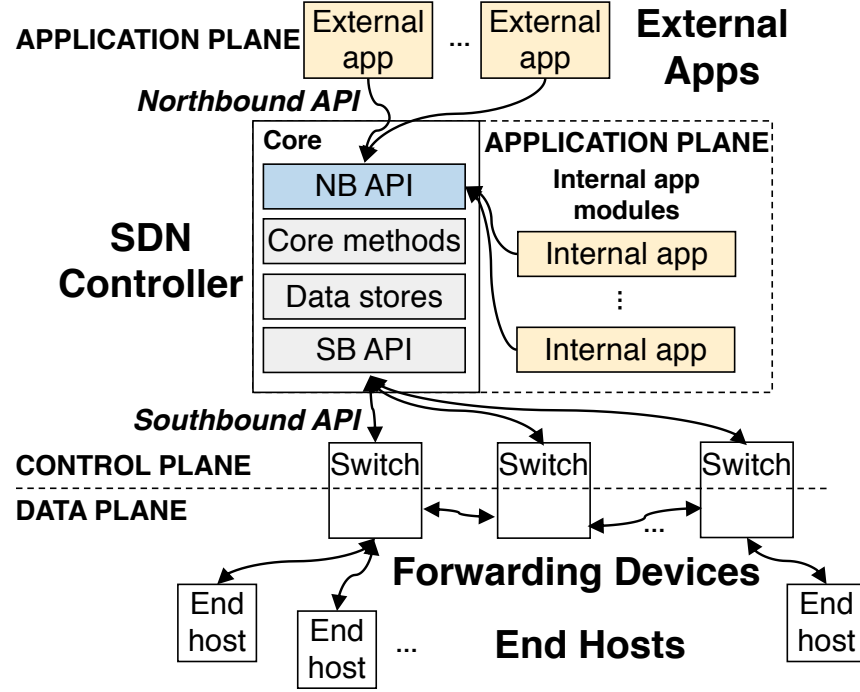
In this chapter, we introduce the software-defined networking (SDN) architecture. We detail the architecture’s various components (Section 2.1), which include the controller, network applications, and forwarding devices. Based on the design of the SDN architecture, we provide a comprehensive overview of the known security-related attacks against the SDN architecture (Section 2.2) and defenses that improve the SDN architecture’s security posture against such attacks (Section 2.3). We categorize attacks related to flooding (Section 2.2.1), information leakage (Section 2.2.2), policy modification (Section 2.2.3), and spoofing (Section 2.2.4). We categorize defenses related to run-time detection (Section 2.3.1), pre-runtime detection (Section 2.3.2), and post-runtime detection (Section 2.3.3).

2.1 SDN Architecture

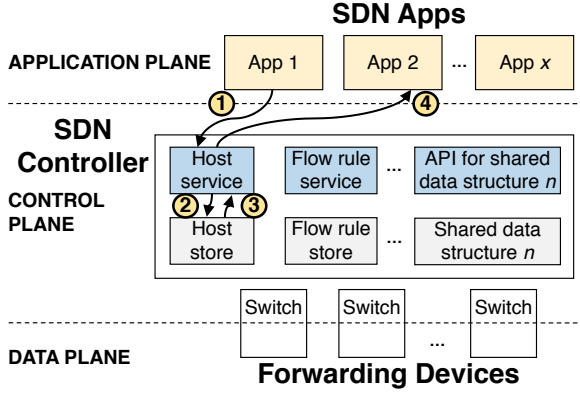
Figure 2.1a shows an overview of the SDN architecture. SDN decouples how traffic decisions are made (*i.e.*, the *control plane*) from the traffic being forwarded (*i.e.*, the *data plane*). Traffic decisions are made in a logically centralized (but perhaps physically distributed) *controller* that functions as the core of a *network operating system*. Controllers manage network configurations and forwarding rules in the network’s forwarding devices through the *southbound API* (*e.g.*, OpenFlow [27]). The *application plane* extends the control plane’s management functionalities through the use of *network applications* (or *apps*) that can query the network’s current state or set high-level intents that influence the network’s state. The aforementioned properties regarding plane decoupling, logical centralization, and programmability distinguish the SDN architecture from traditional networking architectures [1].

2.1.1 Controllers and the Control Plane

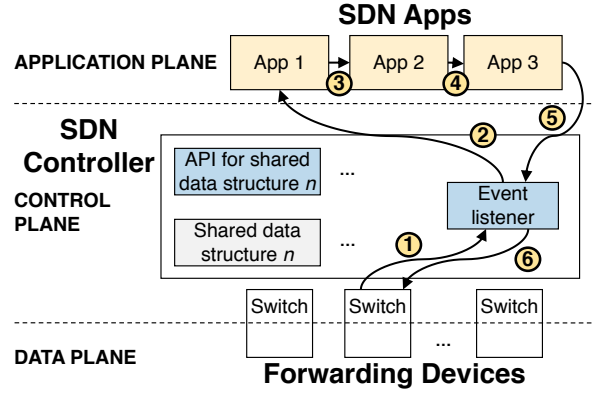
The *SDN controller* (or simply the *controller*) acts as a network operating system to coordinate concurrent applications, to provision resources, and to implement security or network policies [1]. Sev-



(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 2.1: SDN architecture overview and app interactions via service APIs and event callbacks.

eral controller frameworks exist, such as Floodlight [28], Ryu [29], the Open Network Operating System (ONOS) [25], and OpenDaylight (ODL) [30]. Special-purpose controllers for secure envi-

ronments include SE-Floodlight [15], Rosemary [14], and Security-Mode ONOS [19]. Controllers are written in general-purpose programming languages such as Java [15, 19, 25, 28, 30], Python [29], and C [11, 14, 31]. Apps are typically written in the same language as the controller (*i.e.*, as an internal module) but can also be language-independent (*i.e.*, as an external module).

2.1.1.1 Core services, core methods, and data stores

Controllers provide *core services* that maintain the current state of the network and that enable a set of abstractions for additional functionality through apps. For instance, a controller could maintain an abstraction about data plane hosts (*e.g.*, a Host class) that reside on the network (see Section 2.1.1.2 for details about network abstractions). For such host objects, the controller would provide a host core service that enables apps to update information about hosts or retrieve information about hosts. For instance, the ONOS controller provides core services related to hosts, network topology, flow rules, forwarding devices, topology links, topology paths, network configurations, network statistics, and packets [32].

Core services maintain *data stores* that store representations of the network’s state. For distributed controllers that use more than one instance for high availability or fault tolerance, data stores can either be strongly consistent (*i.e.*, network partitions may hinder availability) or eventually consistent (*i.e.*, network partitions will not hinder availability but may induce temporary state inconsistencies across instances). For instance, the ONOS controller uses the Raft protocol for maintaining strongly consistent state [33].

Core services interact with their respective data stores through *core methods*. Apps and forwarding devices interact with the controller’s core services through *application program interfaces* (APIs). (See Section 2.1.1.3 for details about APIs.) Changes to objects within data stores may trigger the firing of event dispatches to event listeners. (See Section 2.1.1.4 for details about events.)

2.1.1.2 Network abstraction model

Controllers expose *abstractions of network objects and processes*. For instance, the ONOS controller includes abstractions for Host objects that represent end hosts, and for Device objects that represent forwarding devices. Those abstractions are built on top of information learned or programmed from lower levels. In ONOS, the host location provider (*i.e.*, a core service) builds Host objects based upon information learned from Packet objects’ header information. As a result, apps that are interested in changes to hosts can reason about such changes at the level of a host abstraction rather than a packet abstraction. That reduces the amount of redundant functionality that would otherwise need to be implemented across various apps.

2.1.1.3 API model

Controllers communicate via application program interfaces (APIs). The *northbound API* allows network applications to query controllers for network state abstractions or set intents about network policy. The *southbound API* allows controllers to set low-level forwarding rules in forwarding devices (e.g., switches and routers) and to query about network state. The *eastbound/westbound API* allows controllers to communicate among themselves or with other SDN systems to exchange distributed state.

Northbound API (controller ↔ app) 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 public API methods. As shown in Figure 2.1b, apps use the host service's `read()` and `write()` methods to interact with the underlying host data store.

There is no standard controller-to-application interface 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 2.1a) or as separate external processes decoupled from the controller. For example, the ONOS controller 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 (controller ↔ forwarding device) SDN controllers also interact with network forwarding devices to disseminate rules and to collect data plane statistics. One popular standard protocol between controllers and forwarding devices is OpenFlow [27]. OpenFlow configures switches' forwarding behavior through flow tables, such that 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 for sending data plane packets to and from the controller (i.e., `PacketIn` and `PacketOut`), to modify forwarding behavior (i.e., `FlowMod`), and to request and receive flow entry statistics (i.e., `StatsRequest` and `StatsReply`), among others. Other southbound API protocols include NETCONF [34], ForCES [35], and P4 [36].

Eastbound/Westbound API (controller ↔ controller) For distributed or federated SDN architectures, controller instances can share relevant state and state changes with each other through an eastbound/westbound API. The underlying consensus protocol for distributed controller instances (e.g., Raft in ONOS [25]) can be viewed as the eastbound/westbound API.

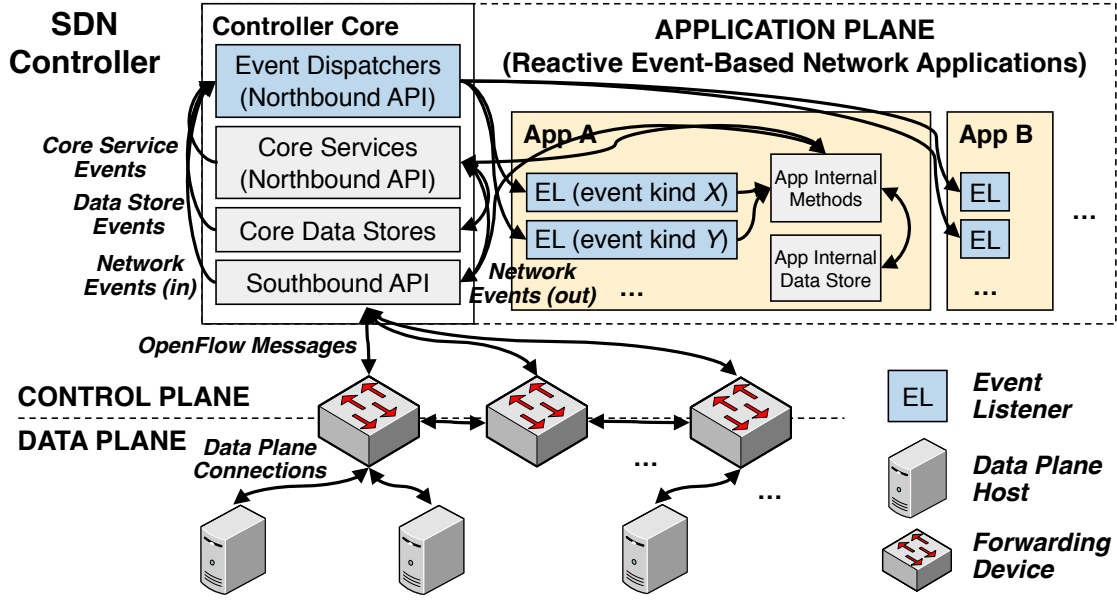


Figure 2.2: SDN architecture with event listeners and event dispatchers. Apps subscribe to event dispatchers and implement event listeners. Network, data store, and service updates generate events.

2.1.1.4 Event model

In addition to API requests, controllers also implement an *event* system. With event callbacks, an app registers itself with the controller to receive events of interest as they occur. In the example shown in Figure 2.1c, 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.

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

Events are sent from *event dispatchers* and received through *event listeners*. Figure 2.2 shows a representative example of an event dispatcher that distributes events to event listeners. For instance, the controller may dispatch a network link event to all apps that are interested in network link state changes (i.e., all apps that have registered link event listeners). An app that cares about new network links can use such an event to make decisions about what functionality to perform (e.g., recalculation of bandwidth for QoS guarantees). Such an app can also gather information about what the control plane's state looks like in the present (i.e., perform API read calls), request changes to the control plane (i.e., perform API write calls), or notify other apps and core services asynchronously

(i.e., perform event dispatching). That process is replicated by other apps and core services that register event listeners and react to events, and the combination of such interactions forms the basis (and complexity) of the event-driven SDN architecture.

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

2.1.2 Apps and the Application Plane

The SDN architecture enables third-party and independent developers to write *network applications*, or *apps*. For instance, HP Enterprise's HPE Virtual Application Networks (VAN) SDN Controller includes an "app store" [37]. The ONOS controller includes a set of reference apps within its core repository [38] as well as a set of sample apps [39]. Such apps enable functionalities such as traffic engineering (e.g., forwarding, routing, and IP address assignment with DHCP), security (e.g., firewall and data plane access control), integration (e.g., OpenStack integration with SDN), monitoring (e.g., telemetry), and general-purpose utilities (e.g., network garbage collection) [38].

As noted earlier, apps can use the controller's API calls and event system to receive updates upon state changes or request the current state. Appendix C.1 provides the skeleton code for a representative internal module app for the ONOS controller. Apps include activation and deactivation methods, event listeners, and API calls.

2.1.3 Forwarding Devices and the Data Plane

The SDN architecture implements the data plane by using a set of *forwarding devices*. In contrast to traditional networking architectures that separate the network into different areas of concern (e.g., Layer 2 forwarding for local networks and Layer 3 routing for wide-area networks), SDN models such switches and routers as generic forwarding devices. Each forwarding device contains a set of *forwarding tables* (or *flow tables*). Each forwarding table contains a set of *forwarding rules* (or *flow rules*) that dictate how packets are matched and forwarded. Each forwarding rule contains a set of *match fields* upon which incoming packets are matched (e.g., IPv4 source and destination prefixes) and a corresponding set of *actions* that can be taken (e.g., drop, forward, or rate-limit).

As SDN centralizes the control plane decision-making in the controller, forwarding devices do not need to make decisions themselves about how to forward traffic. In contrast to traditional networking architectures, each forwarding device can send relevant network configuration packets to

the controller that will make a decision. For example, a controller may send Link-layer Discovery Protocol (LLDP) or Broadcast Domain Discovery Protocol (BDDP) packets into the data plane to learn the network’s topology and link information. Based on that information, the controller may instantiate new forwarding rules and use a southbound API protocol (e.g., OpenFlow) to do so.

As a result of such flexibility in decision-making, forwarding devices can operate much like Layer 2 reactive forwarding switches, Layer 3 routers, or middleboxes such as stateful firewalls. Programmable data planes such as P4 [36] allow for additional flexibility on arbitrary match fields and actions.

Forwarding devices may be implemented in hardware or software. Hardware-based forwarding devices may use ternary content addressable memory (TCAM) to store forwarding tables and forwarding rules for line-rate lookup speeds [27]. Open vSwitch [40] is a popular software-based forwarding device implementation found in network virtualization. Mininet [41] is a network emulator that uses Open vSwitch. OpenFlow-based forwarding devices are uniquely identified through a 64-bit datapath identifier (DPID) that consists of a 48-bit MAC address and a 16-bit implementor-dependent identifier [42] such as a VLAN number.

2.2 SDN Architecture Attacks

The “softwarization” of programmable networks introduces new and varied threats relative to traditional networking architectures. Based on the SDN architecture presented in Section 2.1, we now consider the potential vulnerabilities that the SDN architecture introduces.

Kreutz *et al.* [6] identify seven threat vectors in the SDN architecture: forged or faked traffic flows, switch vulnerabilities, control plane communication vulnerabilities, controller vulnerabilities, trust between controller and management applications, administrative host vulnerabilities, and lack of trusted forensics tools. Hizver [73] proposes a taxonomy for the SDN architecture’s security threats by categorizing threat sources, attacker actions, and threat targets. Benton *et al.* [74] identify OpenFlow-based SDN architecture vulnerabilities. Klöti *et al.* [75] use the STRIDE methodology to identify OpenFlow-based SDN architecture vulnerabilities.

Several surveys investigate the SDN architecture’s security. Scott-Hayward *et al.* [76] categorize SDN security attacks and solutions according to unauthorized access, data leakage, data modification, malicious/compromised apps, denial of service, configuration issues, and system-level SDN issues. Martin *et al.* [77] survey SDN threats based on assets, users, and risks. Khan *et al.* [78] survey SDN topology-based threats. Sattolo *et al.* [79] survey SDN poisoning attacks based on outcomes, layers, and requirements. Yoon *et al.* [80] survey SDN attacks and categorize them based on one of the following attack locations: control plane remote attacks, control plane local attacks, control

Table 2.1: Known Attacks on the SDN Architecture.

Attack Name	Attack Primitive		From	Attacker Target						C/S? MA?	
	What	How		T	I	F	A	M	C		
Data dependency chaining [43]	P	Data dependencies	H	✓	✓	✓	✓	×	✓	×	✓
Cross-plane event vulnerability [44]	P	Data dependencies	H	✓	✓	✓	✓	×	✓	×	✓
Buffered packet hijacking [45]	P	Packet spoofing	A	✓	✓	✓	✓	×	×	×	✓
Crossfire table overflow [46]	F	Flow table capacity	H	×	×	✓	×	✓	×	✓	×
App fingerprinting [47]	I	Fingerprinting	H	×	×	×	✓	×	✓	✓	×
Topology freezing [48]	P	LLDP spoofing	H	✓	×	✓	×	×	×	×	×
Reverse loop [48]	P	LLDP spoofing	H	✓	×	✓	×	×	×	×	×
Babble [49]	F	Packet spoofing	H	×	×	×	✓	×	✓	×	×
Event flooding [50]	F	Event flooding	H	×	×	×	✓	×	✓	×	×
Cross-app poisoning [51]	P	Confused deputy	A	✓	✓	✓	✓	×	×	×	✓
Flow reconfiguration [52]	I	Covert timing	H, S	✓	×	✓	×	×	×	✓	×
Switch identification [52]	I	Covert timing	H, S	×	×	×	×	×	×	✓	×
Out-of-band forwarding [52]	I	Covert storage	H, S	×	×	✓	×	×	×	✓	×
Path reconnaissance [53]	I	CP paths	H	✓	×	✓	×	×	×	✓	×
Table-miss striking [54]	F	Switch table-miss	H	×	×	✓	✓	✓	×	✓	✓
Counter manipulation [54]	F	Switch counter	H	×	×	✓	✓	✓	×	✓	✓
Port probing [55]	P	Race condition	H	×	✓	×	×	×	×	×	✓
Port amnesia [55]	P	State reset	H	×	✓	×	×	×	×	×	✓
Virtual switch vulnerabilities [56]	P	ROP	H	×	×	×	×	×	✓	×	×
Packet injection [57]	S	Packet injection	H	×	✓	×	×	×	×	×	✓
Persona hijacking [58]	P	Multiple protocols	H	×	✓	✓	×	×	×	×	✓
Flow poisoning [58]	P	Race condition	H	×	✓	✓	×	×	×	×	✓
DP saturation [59, 60]	F	Buffer saturation	H	×	×	✓	×	✓	×	×	×
Flow reconnaissance [61]	I	Flow reconnaissance	H	×	×	✓	×	×	×	✓	×
DP intrusion [17]	S	Packet spoofing	A	×	×	✓	×	×	×	×	×
Information leakage [17]	I	CP or DP host	A	✓	✓	✓	×	×	×	✓	×
Rule manipulation [17, 62]	P	Flow manipulation	A, S	×	×	✓	×	×	×	×	×
Attacking other apps [17]	P	App deactivation	A	✓	✓	✓	✓	×	×	×	×
Phantom storm [63]	F	Packet spoofing	H	×	✓	✓	×	×	✓	×	✓
Controller fingerprinting [64, 65, 66]	I	Fingerprinting	H	×	×	×	×	×	✓	✓	×
CP timing [67]	I	Covert timing	H	×	✓	✓	✓	✓	×	✓	×
Host location hijacking [22, 55]	P	ARP spoofing	H	×	✓	✓	×	×	×	×	✓
Link fabrication [22, 68]	P	LLDP injection	H	✓	×	✓	×	×	×	×	×
	P	LLDP relay	H	✓	×	✓	×	×	×	×	×
SDN rootkits [69]	P	Malicious app	C	✓	✓	✓	✓	×	✓	✓	✓
Flow table capacity inference [70]	I	Flow table capacity	H	×	×	✓	×	✓	×	✓	×
Incorrect forwarding [71]	P	Flow manipulation	S	×	×	✓	×	×	×	✓	×
Packet manipulation [71]	S	Packet spoofing	S	×	✓	✓	×	×	×	✓	×
Malicious weight adjustment [71]	P	Group table	S	✓	×	✓	×	×	×	✓	×
Malicious administrator [13]	P	CP configuration	C	✓	×	✓	×	×	✓	×	×
SDN fingerprinting [72]	I	Fingerprinting	H	×	×	×	×	×	✓	✓	×
Dynamic flow tunneling [11]	P	Flow combination	A	×	×	✓	×	×	×	×	✓

Key to attack primitives: F = flooding, I = information leakage, P = policy modification, S = spoofing.

Key to from: H = hosts, A = apps, S = forwarding devices (i.e., switches), C = controller configuration.

Key to attacker targets: T = topology, I = identity, F = forwarding rules, A = apps, M = switch memory, C = controller.

Key to other abbreviations: CP = control plane, DP = data plane, C/S = covert/side, MA = misattribution.

Key to symbols: ✓ = yes, ✗ = no.

channel attacks, and data plane attacks.

We provide an overview of the known attacks against the SDN architecture found in the literature to date. Table 2.1 summarizes the known attacks. We focus on the *attack primitive* that each attack uses, how such a primitive manifests itself, where the attack is launched from, what the intended targets of the attack are, and whether such attacks attempt to use covert channels or misattribute the attack source. Those particular categories for analysis were chosen because they relate to the SDN architecture’s information flow, accountability, and attribution challenges. We organize the attack primitives into flooding (Section 2.2.1), information leakage (Section 2.2.2), policy modification (Section 2.2.3), and spoofing (Section 2.2.4).

We note that certain attack primitives may overlap. For instance, a spoofing attack may induce an unintended policy modification. In such cases, we classify an attack in its own primitive unless it could be considered a subset of another attack primitive. For instance, a spoofing attack is classified as a spoofing attack if its goal is not to induce a policy modification attack. For scoping reasons, we do not include attacks in which data plane hosts attack other hosts through the network, unless specific and unique characteristics of the SDN architecture are involved.

2.2.1 Flooding

Flooding attacks allow an attacker to disrupt the normal operation of the SDN network by causing a degradation or denial of an intended service level. Flooding can cause a system failure when the rate of input to be processed exceeds the processing capacity [24].

Data-plane-to-control-plane saturation Flooding attacks in one component of the SDN architecture can cause systemic effects elsewhere [81]. For instance, flooding in the data plane can cause the control plane to stop functioning correctly, which in turn can cause further data plane denial of service. Shin *et al.* [60] and Ambrosin *et al.* [59] describe the challenges of data plane saturation attacks in which data plane hosts cause control plane denial of service through incomplete TCP handshakes (*e.g.*, SYN flooding) that exhaust available TCAM memory in forwarding devices. Smyth *et al.* [63] describe the “phantom storm” attack, which floods the control and data planes with spurious packets. In that attack, a malicious data plane host uses ARP cache poisoning to send a spoofed IP address into the network. Benign data plane hosts reply, but given that the target host does not exist, the controller broadcasts and floods the data plane with additional packets for each reply. Xu *et al.* [46] propose the crossfire table-overflow attack in which colluding data plane hosts exhaust multiple forwarding tables intermittently.

Control plane reflection attacks use data plane packets as input to trigger expensive control plane decisions. Zhang *et al.* [54] propose the table-miss striking and counter manipulation attacks. Table-

miss striking attacks use targeted packets to determine which match header fields in packets are sensitive to causing such packets to be redirected to the controller (*i.e.*, which packets were a “miss” when matched against a forwarding device’s forwarding tables). Counter manipulation attacks use targeted packets to determine which southbound API messages are sent to the forwarding device.

Distributed controller consistency and event flooding The consistency model assumptions of a distributed controller are also a significant flooding target. Smyth *et al.* [50] show that the targeting of events through event flooding can induce a control plane denial of service when strong consistency guarantees need to be met. Hanmer *et al.* [49] show that the Raft consensus protocol used in ONOS is vulnerable to “babble” attacks, which exploit the timing assumptions about Raft and require expensive recomputations (*e.g.*, network intent recompilation) to be performed in order to flood the controller’s processing capabilities with spurious events.

2.2.2 Information Leakage

Information leakage attacks allow an attacker to infer information about the SDN network’s state without being able to access such information directly. Such attacks can occur through side (or covert) channels that leverage storage or timing information.

Attackers who reside on data plane hosts may not have access to the network’s current state, configuration, and policies. Thus, an attacker may want to know any of the following: 1) whether or not the host resides on an SDN-controlled network; 2) if the network is SDN-controlled, which controller is implemented; and 3) if the network is SDN-controlled, which apps are installed. Such answers allow an attacker to exploit implementation-specific vulnerabilities.

SDN-controlled network fingerprinting Shin and Gu [72] report that an attacker can infer that an SDN architecture is in use (*i.e.*, instead of a traditional networking architecture) because reactive SDN control plane configurations require that the first packet of a flow be processed by the controller. That additional latency is perceptible from the data plane hosts’ perspectives. Bifulco *et al.* [66] show the feasibility of such attacks on hardware-based forwarding devices. Sonchack *et al.* [67] show that the timing differences can be statistically significant in practice.

Controller fingerprinting Azzouni *et al.* [65] fingerprint OpenFlow-based SDN controllers (ODL [30], Floodlight [28], POX [82], Ryu [29], and Beacon [83]) based on timing and packet analysis. Timing-based techniques involve different default values in hard and idle timeouts of flow rules of different controllers, as well as different execution speeds for packet processing. Packet-based

techniques involve different responses to and fields of LLDP and ARP protocol messages for topology and host information discovery. Zhang *et al.* [64] further fingerprint SDN controllers based on specific match fields in packet headers.

App fingerprinting Cao *et al.* [47] use encrypted control plane traffic patterns to infer that specific apps are installed and running in the SDN controller.

Flow rule configuration inference Zhou *et al.* [70] demonstrate that the flow table capacity in forwarding devices can be inferred as a result of the limited amount of memory typically available. Wen *et al.* [17] note that apps can leak sensitive and confidential information about the network configuration to hosts. Liu *et al.* [61] propose flow reconnaissance attacks in which an attacker can force the instantiation of particular flow rules and observe the resulting timings in order to infer existing flow rules that may exist in forwarding devices. Cao *et al.* [53] propose path reconnaissance attacks for inferring whether control plane traffic is being sent in-line with data plane traffic. Thimmaraju *et al.* [52] propose teleportation attacks related to out-of-band forwarding, switch identification, and flow reconfiguration. Such attacks use side channels to relay information about the network's configuration and to exfiltrate information.

2.2.3 Policy Modification

Policy modification attacks allow an attacker to change the desired policy of the SDN network's operation into an unintended state. Such policies include the data plane access control policies (*i.e.*, allowing or denying communication among end hosts), the fairness rules about resource allocation (*e.g.*, network bandwidth capacity), and the control plane access control policies (*i.e.*, allowing or preventing apps from taking certain actions). Given that one of the major design goals of network operating systems is to control and enforce policy, policy modification attacks are of significant concern to the overall SDN security posture.

Policy modification by apps Porras *et al.* [11] show that dynamic flow tunneling attacks can bypass intended data plane access control if the combination of several flow rules is not considered as a group. Furthermore, Wen *et al.* [17] argue that malicious apps can bypass each other's configurations to induce such an attack. They describe how malicious apps can manipulate flow rules if they are given coarse-grained permission to do so and are not detected by other apps. Röpke and Holz [69] propose the notion of SDN rootkits that are implemented through malicious apps. Such rootkits can arbitrarily affect the intended network policy and provide an attacker with remote access. Ujcich *et al.* [51] demonstrate cross-app poisoning in which apps can influence each other's actions through a

shared state control plane model (see Chapter 4). Cao *et al.* [45] further show that buffered packets to be processed by the controller can be manipulated by different apps.

Policy modification by forwarding devices The software processes that execute virtual forwarding devices can be used as attack vectors to influence policy. Thimmaraju *et al.* [56] demonstrate that virtual forwarding devices such as Open vSwitch [40] are vulnerable to return-oriented programming (ROP) attacks. Such attacks can allow an attacker to control the controller process's host. They identify the design characteristics of hypervisor co-location, centralized control, unified packet parsers, and untrusted inputs as key components of the attack surface that can be exploited. Chi *et al.* [71] find that malicious forwarding devices can adjust the weight of traffic and incorrectly forward traffic. Zhang [62] describes how malicious forwarding devices can bypass intended flow rules.

Policy modification by topology modification The modification of the network's topology can indirectly change the network's forwarding policy. Such topology changes arise from changes in network links or in host location.

For changes in network links, Hong *et al.* [22] and Alharbi *et al.* [68] show that unauthenticated LLDP packets for topology discovery can be abused by data plane hosts. Malicious data plane hosts can either generate fake LLDP packets based on well-defined specifications of how such packets should be formulated, or relay LLDP packets in order to create spurious links when seen from the controller's perspective. Such fabricated links can either provide an attacker with eavesdropping capabilities over newly created paths or induce black-hole routing if such links do not actually exist. Port amnesia attacks [55] leverage the dynamic nature of topology changes to give the controller an incorrect view of a port when the port goes down. Marin *et al.* [48] show that the topology freeze attack can prevent the controller from updating its network state when multiple links share the same source port, and the reverse loop attack can force the controller to recompute the network's topology indefinitely from forged LLDP packets.

For changes in host location, Hong *et al.* [22] and Skowyra *et al.* [55] show that data plane hosts can trick the controller into forwarding traffic from benign host locations to attacker-controlled hosts by breaking the MAC-to-port mapping that the controller maintains. Port probing attacks [55] leverage a race condition that exists when a benign host changes its location such that an attacker can attempt to complete the move before the benign host can complete the move.

Jero *et al.* [58] show that attacks against network identifiers can influence the network's topology and forwarding decisions. Persona hijacking attacks [58] break the mappings between the MAC address and the port location, between the MAC address and the IP address, and between the IP address and the hostname. To break the MAC-address-to-port-location binding, the flow poisoning attack [58] leverages a race condition between a victim host and an attacker host in which stale flow

rules temporarily direct data plane traffic to the attacker.

Policy modification by administrators Matsumoto *et al.* [13] note that the centralization of network policy and control in SDN architectures can create administrative challenges if malicious administrators poison the network’s configuration.

Other policy modification Untrusted data plane input can have systemic effects on controller processes and hosts that execute such processes. Ujcich *et al.* [44] demonstrate that cross-plane vulnerabilities can be identified from missing and unhandled events in the controller (see Chapter 5). Xiao *et al.* [43] demonstrate similar vulnerabilities that can execute commands and exfiltrate data from the controller’s host.

2.2.4 Spoofing

Spoofing attacks allow an attacker to mask a data plane sender’s identity or to masquerade as the intended data plane recipient [24]. This attack primitive manifests itself in several SDN components. Chi *et al.* [71] show that compromised SDN forwarding devices can spoof data plane packets that they receive. Even if such packets are encrypted, the forwarding device can cause denial of service through the addition of decryption errors. Wen *et al.* [17] show that compromised apps can use the southbound API to request data plane packets (*i.e.*, PacketIns) or to inject data plane packets (*i.e.*, PacketOuts) arbitrarily, which can induce spoofing by malicious apps. Deng *et al.* [57] show that any unmatched packet handled by forwarding devices can be exploited by data plane hosts to induce false packet processing in the controller.

2.3 SDN Architecture Defenses

We now consider the existing defenses for the SDN architecture against adversarial attacks, and we organize them according to a software development and deployment lifecycle. Such mitigation or prevention of attacks through defenses can occur during the network runtime execution (Section 2.3.1), during the design and testing phases prior to network runtime execution (Section 2.3.2), or during the analysis and forensic phases after network runtime execution (Section 2.3.3). Some overlaps exist between these phases, such as post-runtime analysis that iteratively informs pre-runtime design considerations.

Although the SDN architecture has been proposed as a way to further enhance the security of end hosts in the data plane (*e.g.*, via dynamic and modular access control for end hosts [5]), we

consider such security defenses only if they have systemic impact on the security properties of the SDN architecture or if they target an SDN control plane component.

2.3.1 Runtime Defenses

A *runtime* defense prevents attacks from occurring while the network control plane is executing code. Such defenses necessarily rely on the current runtime conditions in order to mitigate or prevent attacks (*e.g.*, the determination of an access control decision for an app’s API call). Table 2.2 summarizes the runtime defenses for the SDN architecture. We focus on the *defense primitives* in which the defenses map to the attack primitives in Section 2.2. We also organize such defenses in terms of how they are implemented (*e.g.*, access control of APIs), the trusted computing base (TCB) of the components assumed to be secure and trusted, the components from which attacks can be launched, and the components that implement the defense.

Access control of APIs Most access control defenses for the SDN architecture are variants of role-based access control (RBAC). Such defenses implement RBAC in the controller to enforce permissions over the northbound API for app requests. PermOF [10] uses RBAC for a generic SDN architecture and enforces RBAC through a shim layer for access control and a kernel service deputy embedded within the controller. FortNOX [11] extends the NOX controller [31] with signed flow rules and a set of roles for RBAC enforcement. OperationCheckpoint [12] similarly enforces RBAC in Floodlight [28] with a set of read, write, and notification permissions related to OpenFlow commands. SE-Floodlight [15] implements a security enforcement kernel layer that mediates flow rule requests among different roles (*i.e.*, rule chain conflict analysis). ROSEMARY [14] implements a network operating system permissions structure and cites the lack of access control as a challenge for network operating system security. AEGIS [16] checks that apps’ API use is consistent against a set of security policy invariants. SDNShield [17] enforces fine-grained RBAC policies as extensions to controller architectures and implements permission boundary and mutual exclusion policies. SM-ONOS [19] also enforces RBAC with various role granularities; that approach includes bundles (*i.e.*, app or core service), apps (*i.e.*, administrative apps or user apps), or specifically granted permissions.

Dynamic and behavior-based RBAC solutions mitigate the challenges of apps whose behavior may change over time if the apps are compromised by attackers. Controller DAC [18] implements dynamic access control through the use of an intrusion detection system (IDS) that monitors apps’ past northbound API usage. BEAM [20] updates the access control policies based on the dynamic nature of the network.

Ujcich *et al.* [51] note the challenges of RBAC when there are high data dependencies among apps.

Table 2.2: Runtime Defenses for the SDN Architecture.

Defense Name	Defense Primitive		TCB			Attack From				Implementation				
	What	How	C	S	A	H	S	A	C	E	A	C	S	H
FireGuard [46]	F	Rate limiting	✓	✓	✓	✓	×	×	×	×	✓	×	×	×
BEAM [20]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
PROVSDN [51]	P	Access control of APIs	✓	×	×	×	✓	✓	×	✓	✓	×	×	×
SPV [84]	P	Invariant checking	✓	×	×	✓	✓	×	×	×	✓	×	×	×
SWGard [54]	F	Rate limiting	✓	✓	✓	✓	×	×	×	×	✓	×	✓	×
OFMTL-SEC [85]	F	Invariant checking	✓	✓	✓	✓	×	×	×	✓	×	×	✓	✓
(No defense name) [86]	P	Consensus checking	×	✓	×	×	×	✓	✓	×	×	✓	×	×
PacketChecker [57]	P	Invariant checking	✓	✓	✓	✓	×	×	×	✓	×	×	×	×
SECUREBINDER [58]	P	Authentication of messages	✓	✓	✓	✓	×	×	×	✓	×	×	×	✓
SM-ONOS [19]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
LineSwitch [59]	F	Rate limiting	✓	✓	✓	✓	×	×	×	✓	×	×	✓	×
Rule Enforcement [62]	P	Invariant checking	✓	✓	✓	×	✓	×	×	×	✓	×	✓	×
SDN-GUARD [87]	P	Consensus checking	×	✓	×	×	×	✓	✓	×	×	✓	×	×
WedgeTail [88]	I, P	Consensus checking	✓	×	✓	×	✓	×	×	×	✓	✓	×	×
(No defense name) [89]	P	Invariant checking	✓	✓	✓	✓	✓	×	×	—	—	—	—	—
Controller DAC [18]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	✓	×	×
LegoSDN [90]	P	Isolation of apps	✓	✓	×	×	×	✓	×	✓	×	×	×	×
SDNShield [17]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
SDNShield [17]	P	Isolation of apps	✓	✓	×	×	×	✓	×	✓	×	×	×	×
SDNsec [91]	I, P	Invariant checking	✓	✓	×	×	✓	×	×	✓	×	×	✓	×
Timeout proxy [67]	I	Padding	✓	✓	✓	✓	×	×	×	×	×	✓	×	×
JURY [92]	P	Consensus checking	×	✓	✓	×	×	✓	✓	✓	✓	×	×	×
AEGIS [16]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
TopoGuard [22]	P, S	Invariant checking	✓	✓	✓	✓	×	×	×	✓	×	×	×	×
SPHINX [93]	P, S	Invariant checking	✓	×	✓	✓	✓	×	×	×	×	✓	×	×
SE-Floodlight [15]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
(No defense name) [68]	P	Authentication of messages	✓	×	✓	✓	✓	×	×	✓	×	×	×	×
Forwarding Detection [71]	P	Consensus checking	✓	✓	✓	×	✓	×	×	×	✓	×	×	×
Weighting Detection [71]	P	Consensus checking	✓	✓	✓	×	✓	×	×	×	✓	×	×	×
FlowMon [94]	P	Consensus checking	✓	×	✓	×	✓	×	×	—	—	—	—	—
FLOODGUARD [95]	F	Rate limiting	✓	✓	✓	✓	×	×	×	×	✓	✓	×	×
ROSEMARY [14]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
ROSEMARY [14]	P	Isolation of apps	✓	✓	×	×	×	✓	×	✓	×	×	×	×
Fleet [13]	P	Access control of APIs	×	✓	✓	×	×	×	✓	✓	×	×	✓	×
OperationCheckpoint [12]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
AVANT-GUARD [60]	F	Rate limiting	✓	✓	✓	✓	×	×	×	✓	×	×	✓	×
VeriFlow [96]	P, S	Invariant checking	×	✓	×	×	×	✓	✓	×	×	✓	×	×
NetPlumber [97]	P, S	Invariant checking	×	✓	×	×	×	✓	✓	×	×	✓	×	×
FortNOX [11]	P	Access control of APIs	✓	✓	×	×	×	✓	×	✓	×	×	×	×
PermOF [10]	P	Access control of APIs	✓	✓	×	×	×	✓	×	—	—	—	—	—

Key to defense primitives: F = flooding defense, I = information leakage defense, P = policy modification defense, S = spoofing defense.

Key to TCB: C = controller process, S = forwarding devices (*i.e.*, switches), A = apps.

Key to attack from: H = DP hosts, S = switches and/or SB API, A = apps and/or NB API, C = controller configuration.

Key to implementation: E = controller extension, A = app, C = control plane channel, S = switch, H = host.

Key to symbols: ✓ = yes, ✗ = no, — = not applicable.

PROVSDN [51] uses an information-flow-based approach to enforce information flow control (IFC) and prevent “confused deputy” attacks in the SDN control plane (see Chapter 4).

Fleet [13] defends against malicious network administrators who attempt to override policy in the control plane. Fleet implements a switch intelligence layer in the control plane channel between the controller and forwarding devices.

Consensus checking One class of consensus-based defenses uses information from multiple forwarding devices to check that data plane traffic is being forwarded as intended. By computing anomaly scores, FlowMon [94] checks that forwarding device statistics (*e.g.*, port statistics) and forwarding policies agree with each other. Chi *et al.* [71] mitigate malicious forwarding device attacks through weighting and forwarding detection. Those detection algorithms use a consensus of benign forwarding devices to determine whether injected packets are being correctly forwarded and whether injected packets are being forwarded in the ratio that would be expected relative to normal packets. WedgeTail [88] checks forwarding device configurations to determine whether or not packets are being routed according to the intended policy.

Another class of consensus-based defenses checks that controller and app instances are consistent. JURY [92] validates controller responses for distributed controllers that maintain equivalent network views. SDN-GUARD [87] mitigates apps’ ability to act as SDN rootkits by reactively checking that the controller’s and network’s views are consistent with each other. Röpke and Holz [86] propose a similar system that preemptively checks such views for consistency.

A related class of consensus checking involves distributed and fault-tolerant controllers. Insofar as distributed controller instances provide additional redundancy against availability-based attacks, controllers such as ONOS [25], ODL [30], Onix [98], DevoFlow [99], and HyperFlow [100] provide distributed primitives to ensure consensus over the network state. Fault-tolerant controllers use consensus-checking mechanisms to ensure that the network state consistency across controller instances is robust against accidental faults or, under certain sets of assumptions, malicious attacks. SMaRtLight [101] uses a replicated shared data store to maintain the network’s state. Ravana [102] uses a replicated state machine to process control plane events transactionally.

Authentication of messages Several controller architectures, such as FortNOX [11] and SE-Floodlight [15], propose authentication of the northbound API during access control to ensure that apps are authenticated before they perform control plane operations.

TopoGuard [22] and Alharbi *et al.* [68] defend against spoofed LLDP packets through the use of controller-signed LLDP packets that use hash-based message authentication codes (HMACs). SECUREBINDER [58] defends against host-based identity attacks through the use of the 802.1x protocol as a root of trust.

Invariant checking One class of invariant-based defenses checks that the data plane is following the intended control plane policy. NetPlumber [97] checks policy compliance by using a real-time header space analysis [103] approach. VeriFlow [96] ensures that network properties are not violated according to a predefined set of invariants (*e.g.*, loop-free forwarding). SPHINX [93] creates a flow graph view of the network and checks against it for various invariants (*e.g.*, routing rules and path waypoints). SDNsec [91] and Rule Enforcement Verification [62] use the signing of packets by each forwarding device along a path to check that packets forwarded in the data plane are following intended routing policies.

Another class of invariant-based defenses checks that certain control plane actions are semantically valid with regard to normal control plane operation. TopoGuard [22] maintains the state of forwarding device ports (*i.e.*, if a port is a host, a forwarding device, or any device) and checks whether certain state transitions are allowed (*e.g.*, a host port should not be sending LLDP packets). PacketChecker [57] mitigates the packet injection attack by checking an incoming PacketIn packet against MAC address and forwarding device DPID information; further packets are blocked through a flow rule instantiation that prevents further sending of packets to the controller. Smyth *et al.* [89] propose a statistical anomaly approach to vet new links against a baseline of normal behavior for verified links. SPV [84] checks for fake data plane links through the incremental and periodic insertion and verification of probing packets. OFMTL-SEC [85] uses a state machine approach in the data plane to prevent against certain attacks such as ARP spoofing.

Isolation of apps Isolation-based defenses involve the modularization of core services and apps such that malicious components cannot influence the behavior or service levels of benign components. ROSEMARY [14] notes the limitations of monolithic network operating system designs in which all core services operate in a single kernel and mitigates this threat through the logical division of “user-space” and “kernel-space” apps, the monitoring of application resource utilization, and the separation of apps as isolated processes. SDNShield [17] enforces the isolation of different apps via thread sandboxing in Java. Such sandboxing ensures control flow isolation (*i.e.*, an untrusted thread cannot gain trusted privileges), data isolation (*i.e.*, apps cannot reference the controller kernel), and reference monitor remediation (*i.e.*, all system calls must go through a reference monitor). LegoSDN [90] also isolates apps through sandboxes with the goal of being a fault-tolerant controller with recovery mechanisms.

Rate limiting Rate limiting defenses involve proxying, caching, and scheduling techniques. AVANT-GUARD [60] and LineSwitch [59] use variants of proxying for the establishment of TCP connections in the data plane. AVANT-GUARD implements a connection migration module in forwarding devices to prevent saturation attacks from reaching the control plane. However, such a

buffer can cause a bottleneck through denial of service. LineSwitch mitigates the limitations of AVANT-GUARD’s defense through the probabilistic proxying of traffic. FLOODGUARD [95] prevents data-to-control plane saturation by caching packets and using round-robin scheduling to schedule their submission to the controller. SWGuard [54] mitigates control plane reflection attacks by using a multi-queue scheduler to discriminate among southbound API messages. FireGuard [46] mitigates collusion among hosts that attempt to cause data plane saturation attacks.

Padding A timeout proxy [67] defends against timing-based reconnaissance attacks by injecting additional latency into control plane processing. That prevents an attacker from inferring the control plane’s processing load.

2.3.2 Pre-Runtime Defenses

A *pre-runtime* defense removes vulnerabilities in the SDN architecture during the design and testing phases. Such defenses can be either simulated without executing network code (*e.g.*, static analysis), or executed using a testbed framework (*e.g.*, fuzzing).

Model-based Model-based defenses use a set of invariants to check that a network is operating according to well-known correctness properties (*e.g.*, the assurance of a loop-free data plane topology). Anteater [104] translates network invariant properties and the data plane state into Boolean satisfiability (SAT) problems. NICE [105] uses model checking and concolic execution against a set of network correctness properties to find invalid possible controller states. VeriCon [106] verifies network correctness properties against admissible topology configurations by using an SMT solver. SDNRacer [107] models the happens-before relationships for data plane configuration updates. Attendre [108] checks for data plane race conditions that may be present between control plane configuration changes and data plane packet forwarding. CONGUARD [109] checks for control plane time-of-check-to-time-of-use (TOCTTOU) race conditions and uses a happens-before model of the control plane’s app lifecycles and event notifications.

Fuzzing-based Fuzzing-based defenses use invalid input to determine whether the SDN architecture is behaving in an undesired manner. STS [110] uses a delta debugging technique to produce the minimal causal sequences necessary to reproduce a bug. BEADS [111] automates fuzz testing for OpenFlow-based forwarding devices that do not follow the OpenFlow specification [42]. ATTAIN [81] provides fuzzing capabilities as part of an attack description language within an OpenFlow-based attack injection framework. DELTA [112] provides a security assessment framework for testing controller implementations against common control plane attacks.

Program-analysis-based Program-analysis-based defenses use software program analysis techniques, such as static analysis, to analyze the programmable code to be executed in the network control plane. SHIELD [113] and INDAGO [114] use static analysis over apps to generate behavioral profiles of apps' API calls and to determine which apps misuse those calls. EVENTSCOPE [44] uses static analysis to summarize event flow and API use among core services and apps and identifies where unhandled events may cause vulnerabilities (see Chapter 5). SVHunter [43] uses static analysis to show how data dependencies from data plane input can maliciously influence the controller and the controller's host to execute remote code.

Secure-by-construction Secure-by-construction defenses use safe language constructs to remove certain vulnerabilities by design. Frenetic [115], Pyretic [116], and NetCore [117] use a declarative-based language to provide composable and modular app functionality. Merlin [118] provides a language for network resource requests that translates those requests into constraint satisfaction problems. NetKAT [119] provides a language based on Kleene algebra that ensures composability and isolation properties. Flowlog [120] consolidates control and data plane abstractions and provides a limited-expressiveness network programming language that is amenable to verification. Cocoon [121] provides a language for high-level network intent specification and verifies intents with a stepwise refinement approach.

2.3.3 Post-Runtime Defenses

A *post-runtime* defense provides insight into attacks after they have occurred. Such defenses are necessary for auditing and accountability [122] (see Chapter 3), and they may influence design decisions about what properties should be verified during pre-runtime phases (see Section 2.3.2).

Network debugging and troubleshooting Network debugging tools are used to find the sources of bugs and errors but can also be used to pinpoint attacks. OFRewind [123] provides recording and replaying capabilities for OpenFlow-based SDN architectures. OFRewind captures user-specified subsets of control and data plane traffic during network execution. Users can replay that traffic later for troubleshooting and debugging purposes. NetSight [124] provides a packet history of how packets traversed the data plane, along with querying and filtering capabilities. NDB [125] allows users to set packet backtraces and to set breakpoints for debugging purposes in the data plane. BigBug [126] clusters concurrency violations of expected network behavior. Net2Text [127] summarizes network activity using natural language processing techniques. FALCON [128] provides fault localization for the control plane based on a baseline of expected behavior.

Network forensics and provenance Although network debugging and troubleshooting tools can provide some insight into attacks after they have taken place, other tools have been developed that focus on the causal dependencies within the SDN control and data planes. For network-operating-system-like controllers, FORENGUARD [129] captures data dependencies in the control plane and stores such dependencies graphically. PICOSDN (see Chapter 6) similarly captures dependencies graphically for backward and forward tracing, and also provides a data plane model to show the data plane’s causal role in reactive control planes. For declarative controllers, network provenance tools [130, 131, 132] can aid in causal explanations and, under certain conditions, can help determine how a network configuration should be repaired.

CHAPTER 3

ACCOUNTABILITY IN SOFTWARE-DEFINED NETWORKING

Software-defined networking (SDN) overcomes many limitations of traditional networking architectures because of its programmable and flexible nature. Security applications, for instance, can dynamically reprogram a network to respond to ongoing threats in real time. However, the same flexibility also creates risk, since it can be used against the network. Current SDN architectures potentially allow adversaries to disrupt one or more SDN system components and to hide their actions in doing so. That makes assurance and reasoning about past network events more difficult, if not impossible. We argue that an SDN architecture must incorporate various notions of accountability for achieving systemwide cyber resiliency goals. We analyze accountability based on a conceptual framework, and we identify how that analysis fits in with the SDN architecture’s entities and processes. We further consider a case study in which accountability is necessary for SDN network applications, and we discuss the limits of current approaches.

3.1 Introduction

Software-defined networking (SDN) has emerged as a new networking architecture that attempts to overcome some of the limitations of traditional networking. SDN is distinguished by a logically centralized but physically distributed programmable control plane in which decisions about forwarding are decoupled from the traffic being forwarded [1]. This flexibility has encouraged SDN adoption in enterprise, campus, cloud, mobile, and telecommunication networks, among others [1].

At first glance, SDN enhances the ability of security services to protect the network’s end hosts from threats. The architecture’s global perspective [3] allows the control plane to monitor traffic flows and abstract such information for network security applications’ use. Further, the architecture’s programmable nature allows administrators or security applications to rapidly reconfigure the network’s forwarding behavior to adjust to threats in real time.

However, such insight and flexibility are not without costs, as the increased attack surface and centralized programmatic control could be used against operators and administrators. If we assume that an attacker has already compromised an SDN system’s components to control network behavior and that the attacker lies, equivocates, and hides or destroys evidence of such actions [69], how can

Table 3.1: Summary of Design Considerations and Properties for SDN Accountability.

Who is accountable to whom	What one is accountable for	Assurance mechanisms	Standards	Effects of breach
<ul style="list-style-type: none"> • Software process level <ul style="list-style-type: none"> – Switch–switch – Controller–switch – Controller–application – Controller–controller • User level <ul style="list-style-type: none"> – Network administrators – Security administrators – End users • Organizational level <ul style="list-style-type: none"> – Clients–providers – Peers 	<ul style="list-style-type: none"> • Forwarding / topology • Intent / policy <ul style="list-style-type: none"> – Network resources – Constraints – Criteria – Instructions • Configuration • Authorization / access <ul style="list-style-type: none"> – Permissions and roles – Authentication and access 	<ul style="list-style-type: none"> • Data provenance • Authenticated logging <ul style="list-style-type: none"> – Tamper-proof – Non-repudiable • Fault tolerance <ul style="list-style-type: none"> – Byzantine fault tolerance – Graphical modeling – Blockchains • Roots of trust 	<ul style="list-style-type: none"> • Legal • Regulatory • Policy • Contractual 	<ul style="list-style-type: none"> • Deterrence <ul style="list-style-type: none"> – Loss of money – Loss of reputation • Resiliency <ul style="list-style-type: none"> – Response – Recovery

we be assured of past event integrity? Furthermore, how can we trust the provenance of such events, attribute them to the responsible entities, and take actions against them to hold them responsible and meet systemwide cyber resiliency goals?

We argue that *accountability by design* [133, 134] is necessary for SDN. As SDN architectures become increasingly complex distributed network operating systems [25], we see a need for ensuring accountable practices at multiple levels among various entities and stakeholders. Achieving accountability is not strictly a technical problem—legal, regulatory, and policy frameworks all help guide accountable systems design—but accountability requires data assurances to correctly identify responsible entities when taking responsive actions to support resiliency goals.

Our contributions include application of a conceptual framework of accountability [135] to SDN to identify the agents, system entities, processes, and standards involved in network accountability assurance. We find that while previous work has considered some of these aspects, no complete design solution exists today that systematically incorporates all of them. We illustrate the need for accountability through a practical case study scenario of SDN network applications.

3.2 Designing an Accountable SDN Architecture

Accountability is the “security goal that generates the requirement for actions of an entity to be traced uniquely to that entity” and which supports “nonrepudiation, deterrence, fault isolation, intrusion detection and prevention, and after-action recovery and legal action” [24]. We borrow concepts from the public policy domain, in particular the accountability framework for designing accountable systems proposed by Mashaw [135], to analyze accountability as it applies to SDN architectures and entities. Table 3.1 summarizes the design considerations and desirable properties for SDN accountability.

3.2.1 Who Is Accountable to Whom?

We follow Mashaw in describing accountability as relationships between entities. To say “*A* is accountable to *B*” means that *A* behaves according to processes guided by standards by which *B* can correctly attribute *A*’s actions and take responses against *A* if *A* deviates from such processes and standards. (The meanings of *processes* and *standards* are explained in detail throughout the remainder of Section 3.2.)

We organize those relationships at three levels: software process, user, and organizational accountability.

3.2.1.1 Software process level

SDN software components should keep each other accountable for low-level network state changes so as to attribute actions to particular software instances for troubleshooting (*e.g.*, fault isolation) or for forensics. We identify the following accountability-critical classes of inter-process relationships:

Switch–switch Switches should keep each other accountable for their data plane forwarding actions. In particular, they should ensure that packets traverse the correct switches to enforce isolation guarantees and forwarding accountability [91].

Controller–switch Controllers should keep switches accountable for their actions to ensure that network intents are followed. For instance, switches should attest to their current forwarding behavior state and report it to controllers.

Controller–application Controllers should keep network applications accountable to ensure that conflicting policies are mitigated according to a permissions model [15]. To ensure trustworthy network applications [6], it is necessary to hold network applications accountable for actions they take

that affect the network state [15]. Application developers and publishers should be held accountable, too.

Controller–controller In contrast to a single centralized controller, distributed controllers should provide high availability, scalability, and fault tolerance properties [25]. Distributed controller instances share copies of the network state and may act as clients in reading from a distributed data store [101, 102]. Given that the network’s intelligence is logically centralized in the controllers, they should keep each other accountable for network state changes.

3.2.1.2 User level

Network and security administrators should keep each other accountable for decisions that affect network state, particularly if the administrators have the potential to collude or are assumed to be individually untrusted [13]. Administrators should keep the network’s end users accountable for their actions on the network, such as equitably sharing network resources based on policy.

3.2.1.3 Organizational level

Organizations should keep other organizations accountable for network resources when considering client–provider or peer models. In cloud computing, for instance, a provider should use a telemetry service to account for network resources used by clients, and clients should be able to audit providers to ensure that the services requested are being provided in practice [136]. For an autonomous system (AS), each AS should make other ASes accountable for their Border Gateway Protocol (BGP) advertisements or for the inter-AS network resources (*e.g.*, bandwidth) they use related to peering agreements.

3.2.2 What Is One Accountable For?

An accountable architecture accounts for the system’s “state” and state changes via events or actions taken by system entities. Here, we identify several notions of state.

Forwarding behavior and topology From the data plane perspective, the network’s state consists of the forwarding behavior (*e.g.*, flow table entries) and the topology (*e.g.*, ports, links, switches, hosts). In OpenFlow-based SDNs, the forwarding behavior is defined by flow tables that consist of flow entries with matching attributes and a set of actions to take for matching packets [42]. The

topological information is based on switch configuration [42] and also from auxiliary protocols such as the Link Layer Discovery Protocol (LLDP) and the Address Resolution Protocol (ARP) [28].

Intent and policy From the application plane perspective, the network’s state consists of the policies implemented by intents. ONOS [25], for instance, defines intents by network resources (*e.g.*, ports, links), constraints (*e.g.*, bandwidth), criteria (*e.g.*, matching headers), and instructions (*e.g.*, header modifications, output). Intents contrast with specific protocols like OpenFlow [42] by abstracting the implementation details.

Configuration From a system administrator’s perspective, each network component requires configuration. OF-Config [137] configures switches, changes port states, and changes security certificates, among other functions.

Authorization and access From a security administrator’s perspective, the network requires a system for authorizing users’ actions based on roles and permissions. Such a system needs to record state modification, permissions, and authentication and access events, among other records [15].

3.2.3 What Process Assures Accountability Mechanisms?

We now consider four necessary classes of assurance mechanisms, including their uses to date in SDN and other fields.

Data provenance Arguably, the most important property of data assurances for accountability is their ability to answer questions about where data came from and why the data came to be [138]. Data provenance answers these questions by attributing data to their sources in order to support audit trail generation [139]. Data provenance has been used in database systems [138] and in distributed systems for identifying which system components took specific actions [140]. Dwaraki *et al.* [141] model SDN forwarding state changes through a distributed version control system to answer provenance queries about network state, though the architecture does not assume an adversarial setting.

Authenticated logging While data provenance explains data origins, an adversarial setting requires assurance that the stored data cannot be tampered with. Authenticated data structures (*e.g.*, Merkle trees) provide a way of implementing tamper-proof logs [142]. Each log entry is associated with a cryptographic hash, and tampering with previous log entries makes tampering evident. Among different entities, each entity can digitally sign entries it makes to the log and thus cannot

repudiate previous entries it has signed. Porras *et al.* [15] extend the Floodlight SDN controller to include an application audit module for associating logged events with their sources.

Fault tolerance Accountability does not begin only after the system as a whole has already failed. Many systems are designed to tolerate (or mask) failed components. When some component fails, the failed component should be held accountable for its actions and for events attributed to it.

Byzantine fault tolerant (BFT) protocols are a practical way to make many systems more robust, and the security they offer is evaluated by the strength of their guarantees (*i.e.*, the weakness of their assumptions). PBFT [143], for instance, guarantees that a network of $N = 3f + 1$ replicas can tolerate up to f corrupted instances that behave arbitrarily or maliciously. Furthermore, most BFT protocols guarantee liveness and high availability under very weak assumptions about the ability of the uncorrupted nodes to communicate; they also guarantee consistency even in a completely asynchronous network.

While the system as a whole should exhibit some degree of fault tolerance (as previously explored in the SDN context in [101, 102]), the subsystems used to ensure accountability should be especially fault-tolerant. BFT protocols must typically rely on at least a majority of the nodes to ensure safety and liveness, though secure network provenance (SNP) can rely on an even weaker assumption [144]. SNP uses a provenance graph to capture events in a distributed system, and minimally requires only one correct node to have witnessed an event in order to attribute it. Cryptocurrencies have recently popularized the use of widely distributed BFT protocols to provide a transparent and publicly verifiable transaction log known as a “blockchain.” Blockchain updates are relatively expensive and slow, but this trade-off may be appropriate for accountability-critical information.

Roots of trust By adopting accountability as an explicit design goal, we strive to reduce the amount of trust in the system. However, no design is perfect, and we believe practical architectures will still require some “roots of trust” upon which the system’s security relies. Explicit descriptions of these roots of trust will be essential to evaluating accountability designs. To validate a design, we must identify the trusted entities and justify their trustworthiness. In the SDN context, for instance, Jacquin *et al.* [145] propose a trusted SDN architecture by placing trust in trusted platform modules (TPMs).

3.2.4 By What Standards Should Accountability Be Judged?

Given that the network infrastructure is a central component of many institutions, and that it can “see” everything (including sensitive data) [3], an accountable SDN may be necessary in practice for

meeting or aiding legal, regulatory, policy, or contractual requirements. In this context, standards set the accountability requirements that must be met.

Accountability standards are derived through laws, regulations, and policies. In the U.S., for instance, federal laws and regulations set domain-specific accountability standards as they apply to health records (*e.g.*, HIPAA), educational records (*e.g.*, FERPA), and financial records (*e.g.*, Gramm–Leach–Bliley Act), among other domains.

We can also apply accountability in the context of other policies, such as network neutrality. The Council of the European Union recently passed network neutrality regulations for European Union member states, noting that “a significant number of end-users are affected by traffic management practices which block or slow down specific applications or services” [146]. Here, accountability includes customers and regulators who keep network providers accountable for their network management practices, as the regulation states that “reasonable traffic management measures... should be transparent, non-discriminatory and proportionate, and should not be based on commercial considerations” [146]. Accountable designs can help ensure compliance with these regulations and can support their enforcement.

Outside of established legal regulations, any two parties can decide to enter into contractual service level agreements (SLA) regarding network resources, and the agreement’s terms can set the standards that determine which entities and processes assure accountability and the effects of breaching the standards.

3.2.5 What Are the Effects of Breaching Standards?

Accountability can provide a natural deterrent against some classes of attacks, and can therefore have a passive effect of helping parties conform to agreed-upon standards. However, accountability can also play a more active role in system resiliency by supporting responses that restore the system to correct function after a failure.

Deterrence An accountable SDN architecture may provide a disincentive to attack the network or deviate from agreed-upon rules, as such an attack or deviation could be attributed to the responsible entity [147]. As a result, the responsible entity has something to lose if it had previously pledged something of value in order to participate [148]. The loss may be monetary, as detailed by an agreed-upon SLA or smart contract [149]. Alternatively, the loss may be implicit and reputational, such that other entities choose not to participate with the responsible entity after discovery [148].

Resiliency Such deterrence alone may not provide enough disincentive to stop an attacker from attacking a network, and in such cases, intrusion tolerance designed to meet system resiliency goals

is necessary. The resiliency process is often defined as comprising detection, response, and recovery phases. Accountability clearly plays a role in detection, but it also provides essential information for effective responses.

Each entity may audit other entities to identify misbehavior and attribute it to the responsible entity. Upon detection, an entity may report to other entities to indicate that they should take some response action. For instance, a misbehaving entity might be isolated by peer entities so as to allow for human intervention (*e.g.*, forensic analysis by a security administrator) while still meeting system service goals. Finally, for recovery, two existing mechanisms for SDN include partial configuration rollback [150] and elastic controller provisioning [151].

3.3 Case Study: Accountable SDN Applications

We now consider a case study of applying the accountability process to SDN network applications.

3.3.1 Scenario

As controller software has become increasingly complex, there has been a proliferation of available network applications (apps) that network and security administrators can deploy. HP Enterprise, for instance, offers an SDN App Store [152] where users can download monitoring, security, optimization, orchestration, and visualization tools that coordinate with the HP VAN SDN controller. At the time of this writing, 35 of the 39 apps are provided for free, and 29 of the 39 apps were developed by third parties outside of HP Enterprise [152].

A natural security question arises: How can we trust network applications? Furthermore, how can we attribute actions that they take? Consider a simple example of three systemwide network applications used by a cloud provider: 1) an intrusion detection system (IDS) app monitoring all external-bound data plane traffic for intrusions, 2) a quality of service (QoS) app supporting network SLAs between the cloud provider and its clients, and 3) a firewall app protecting the data plane from external threats and isolating inter-client traffic.

Suppose the IDS app discovers a potential intrusion with systemic consequences in one of the client's resources, and the "discovery" is later determined to be a false positive. While the potential intrusion is still considered a real threat, the firewall receives the IDS alert and reactively reconfigures the network to block traffic that affects other clients' network resources. The QoS app determines it cannot provide any routing paths that support other clients' SLAs now that particular routes have been blocked, and the agreed terms from the SLAs are thus breached as a result of a nonexistent threat.

Each app, viewed independently, provided its respective services correctly, yet the actions taken on behalf of one client negatively affected other clients. Which entities should be held accountable for breaching the SLAs—the client whose resources caused the alert to be generated, the cloud provider whose security policies required IDS monitoring, the apps’ developers whose software generated alerts and actuated the responses that breached the SLAs, or a combination thereof?

Porrás *et al.* [15] consider the application coexistence problem as one in which multiple applications compete to make decisions that affect network behavior. They propose a mediation policy that includes minimal permissions levels, and they suggest implementing application accountability through a security audit service module. While their auditing solution attributes events to the applications that generated them, we posit that this is only one component in the accountability process. How the data can be used afterward to provide attribution, how the collected data relate to each other, and how the data can be used to enforce automatic penalties for the accountable entities must also be considered.

3.3.2 Analysis

Our framework, described previously in Section 3.2, moves accountability from the view of what data are collected to a complete end-to-end view of how those data can be used to assign attribution and drive responsive decisions automatically. We highlight parts of the three-application scenario in which the conceptual framework of accountability can help in the SDN architecture design process.

Provider and clients The cloud provider is accountable to its clients for providing acceptable levels of network service (*e.g.*, bandwidth, latency, denial of service protections) as agreed upon contractually in an SLA. Its clients should be able to audit the provider to ensure that the service is provided in practice [136]. The provider and clients can agree on culpability when service levels are breached, and they can codify this logic with smart contracts that include monetary stakes. If a client determines that service is not being provided, the client can receive monetary compensation [149].

Controllers and apps The three apps are accountable to the SDN controllers for the high-level intents that they ask the controllers to implement. The SDN controllers are accountable to the apps for the low-level actions they take in response, as apps may require assurance that certain actions were taken or to provide evidence if disputes arise later. For instance, if the QoS app cannot change the network’s routing to meet the SLA because of an action that the firewall app requested, the QoS app can use evidence of the firewall app’s actions to declare its innocence as the root cause of the SLA failure. (The firewall app may do the same to declare its innocence as the root cause of failure vis-à-vis the IDS app.)

3.3.3 Remarks

Based on our accountability framework’s considerations, SDN has made some progress in provenance [141], secure auditing [15], fault tolerance [101, 102], roots of trust [145], and resiliency [150, 151]. However, no one design captures *all* of the elements required for an accountable architecture. As illustrated in our case study, no solution to date has considered accountability as an end-to-end process, starting from assured data guarantees, continuing with auditing and detection of breaches, and ending with automated actions for deterrence and response that support resiliency goals.

3.4 Conclusion

SDN continues to be applied in a multitude of enterprises and domains [1], and its global perspective [153] can aid in providing detection and response mechanisms for systemwide cyber resiliency. We argued that the security property of accountability must be considered in the architecture design so as to support detection assurances that ultimately inform responses. We provided a conceptual framework analysis of accountability as applied to SDN entities and processes, and we applied several notions of accountability in our network application case study. We hope that this study spurs further interest in incorporating accountable networking by design.

CHAPTER 4

CONTROL PLANE CROSS-APP POISONING

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.

4.1 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 [154]. The rapid growth and potential value¹ 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 [37].

¹A 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 [154].

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 [6, 76, 155, 156]. 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” [156].

To date, defenses that limit the SDN attack surface have included app sandboxing [14], TLS-enabled APIs [15, 25, 30], API abuse prevention [16, 18], and role-based access control (RBAC) for apps [15, 19], 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.

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 [155], leading to a “confused deputy” problem [157]. 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 [158]. 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.

Overview 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 [25] 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 [19]. 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.

Contributions 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.

Organization This chapter is organized as follows. In Section 4.2, we outline the threat model depicting our attacker’s capabilities and goals. In Section 4.3, we provide an overview of information

flow challenges in the SDN control plane. In Section 4.4, we present our methodology for detecting CAP attacks. In Section 4.5, we show CAP attacks’ existence using Security-Mode ONOS as a case study. In Section 4.6, we outline IFC policies to counteract CAP attacks. In Section 4.7, we present the design, implementation, and evaluation of our defense, ProvSDN. In Section 4.8, we discuss challenges and design trade-offs, and in Section 4.9, we discuss related work. We conclude in Section 4.10.

4.2 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 that do not implement RBAC make it trivially easy for apps to modify and poison data that other apps use. Lee *et al.* [112] 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 unsubsubscription. 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

²For 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 [159]. As of August 2018, ONOS has also been issued 12 CVE entries, including arbitrary apps being loaded into the controller [160].

³Aruba Networks, a subsidiary of Hewlett-Packard Enterprise, maintains an “SDN app store” for the HP SDN controller [37]. As of August 2018, the app store contained 12 apps from third-party developers and 13 apps from “Aruba Technology partners.”

be enforced using a public key infrastructure (PKI) for authentication [6], and several controllers (e.g., [15]) already do so.

4.3 Challenges

We provide a brief overview of information flow, the SDN control plane’s information flow challenges and our main contributions in solving such challenges.

4.3.1 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 [158]. Pasquier *et al.* [161] provide an overview of classical information flow models. Among them is one proposed by Biba [162], 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 receiver, which implies that low-integrity information cannot reach high-integrity subjects. Myers and Liskov [163] relax the hierarchical assumptions by proposing a system of integrity tags and labels assigned to subjects.

4.3.2 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.

4.3.2.1 Lack of well-defined application isolation and enforcement as applied to shared control plane state

Some controllers, such as Rosemary [14], sandbox each app’s resources (*e.g.*, memory and CPU usage) and use RBAC to allow apps or prevent them from 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 [158]. 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 4.4, and demonstrate its consequences through an attack evaluation in Section 4.5.

4.3.2.2 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 [6, 156] or to quickly narrow down and attribute blame to specific apps when something goes wrong [164]. This lack of insight could mislead practitioners into incorrect conclusions when they investigate their systems.

Our contributions In Section 4.4, we describe how to use a cross-app information flow graph to better understand the attack surface. In Section 4.7, we show how data provenance can provide insight into enforcement and recording of control plane activities.

4.4 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 4.4.1–4.4.3), and we overlay this model with apps’ actual data flows (Section 4.4.4). While individual examples of CAP attacks have been considered in the SDN security literature (e.g., [155]), 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.

4.4.1 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 $A = \{a_1, a_2, \dots, a_x\}$, that comprise the apps in the SDN application plane.
- A set of roles, denoted by $R = \{r_1, r_2, \dots, r_y\}$.
- A set of objects, denoted by $O = \{o_1, o_2, \dots, o_z\}$, 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 SDNShield [17] specifies objects at the sub-API granularity (*e.g.*, read flow tables with a specific IP prefix).

4.4.2 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 [165].

Algorithm 4.1 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  $(r_j, p_j) \in m_{RP}$  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:       end if
7:       if  $p_k \in P_W$  then
8:          $\mathcal{E} \leftarrow \mathcal{E} \cup \{(a_i, o_k)\}$ 
9:       end if
10:    end for
11:  end for
12: end for
13:  $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E})$ 
```

Algorithm 4.1 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.

4.4.3 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 \mathcal{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 4.1, the path nodes alternate between apps and objects. We define $\mathcal{C}_v(\mathcal{G}) = \langle a_0, o_1, a_2, \dots, a_{n-1}, o_n \rangle \mid n \geq 3; n \text{ is odd}$.

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 \mathcal{C}_v path are the apps that app a_0 can co-opt

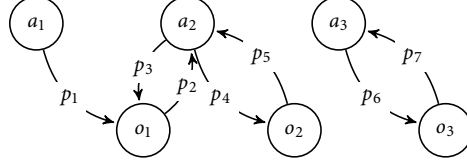


Figure 4.1: Example of a cross-app information flow graph \mathcal{G} with a cross-app attack vector $\mathcal{C}_1 = \langle a_1, o_1, a_2, o_2 \rangle$. App a_1 may be able to poison object o_2 even though it does not possess permission p_4 to do so; instead, it would use object o_1 , app a_2 , and app a_2 's permission p_4 . App a_1 cannot poison object o_3 , since no path exists between them.

using permissions that a_0 itself does not possess. Similarly, any intermediate objects in a given \mathcal{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.

Consider the example cross-app information flow graph in Figure 4.1. Continuing the example from Section 4.1, 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 .

4.4.4 Cross-App Poisoning Gadgets

Our methodology in Sections 4.4.1–4.4.3 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 4.5.

4.5 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 [19, 25]. We chose the ONOS framework because it is a representative example of a popular, production-quality controller used in industry by telecommunication service providers [159], among others. The ONOS framework is Java-based with publicly available source code⁴ bundled with open-sourced apps. Security-Mode ONOS is a variant of the ONOS SDN controller with additional support for RBAC.

4.5.1 CAP Model for Security-Mode ONOS

4.5.1.1 Apps

The v1.10.0 release includes 64 bundled reference apps [38] 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).

4.5.1.2 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 chapter, 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.

4.5.1.3 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

⁴Throughout the chapter, we use the ONOS v1.10.0 source code available at [166].

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.

4.5.1.4 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.

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 B.1 in Appendix B.1.2 for additional details.)

4.5.1.5 Cross-app information flow graph

Using the security reference policy, we applied Algorithm 4.1 to generate the cross-app information flow graph \mathcal{G}_{ONOS} for ONOS with all apps included.⁵ Figure 4.2 shows the complete \mathcal{G}_{ONOS} with 88

⁵We 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.

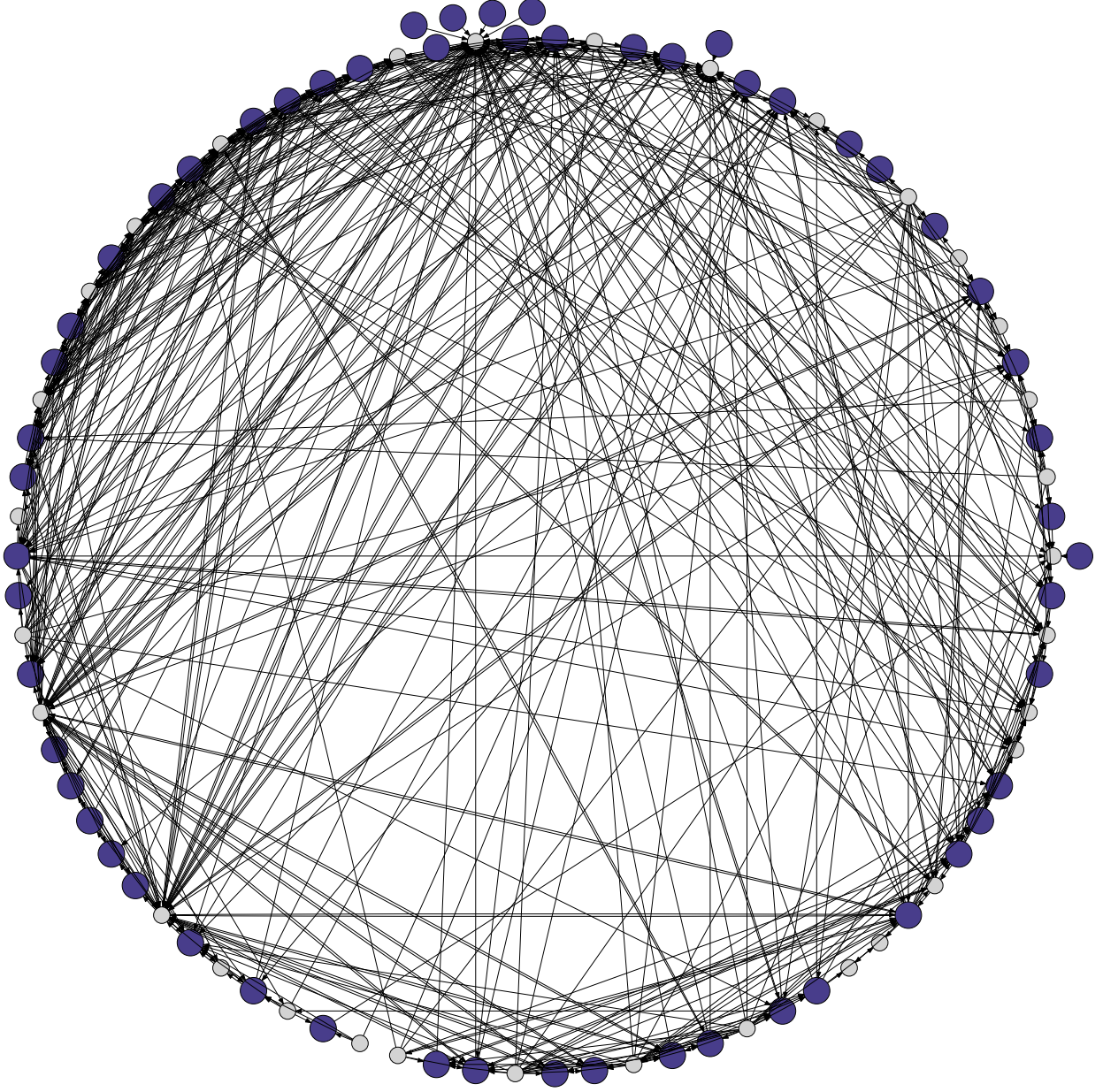


Figure 4.2: Cross-app information flow graph \mathcal{G}_{ONOS} using the 64 apps included with ONOS. Large points represent apps; small points represent objects in the shared SDN control plane state; and arrows represent permissions for apps to read from or write to objects.

nodes⁶ 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 4.3) and how many apps each object could be accessed by, either directly or indirectly (Figure 4.4).⁷ For both analyses, we removed an app named

⁶Manager classes whose methods were not called by any app were not included in the cross-app information flow graph; thus, $|A| + |O| \neq 88$.

⁷A shortest path in \mathcal{G} of length 3, for instance, corresponds to indirect accessibility via 1 app in Figure 4.3 or 1 object in Figure 4.4.

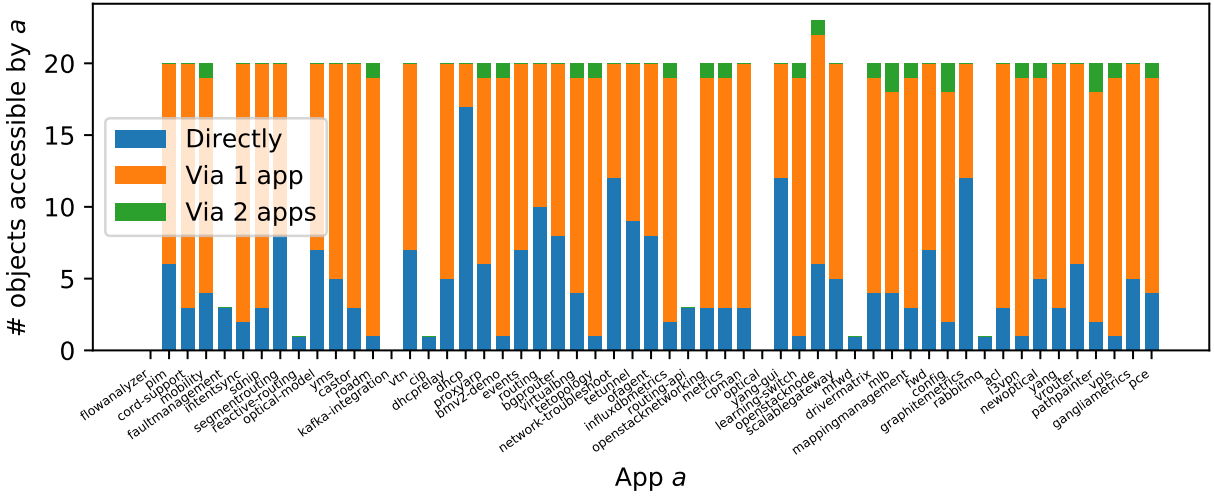


Figure 4.3: App to object accessibility (via shortest paths) in \mathcal{G}_{ONOS} with 63 apps. Paths begin at a given app a .

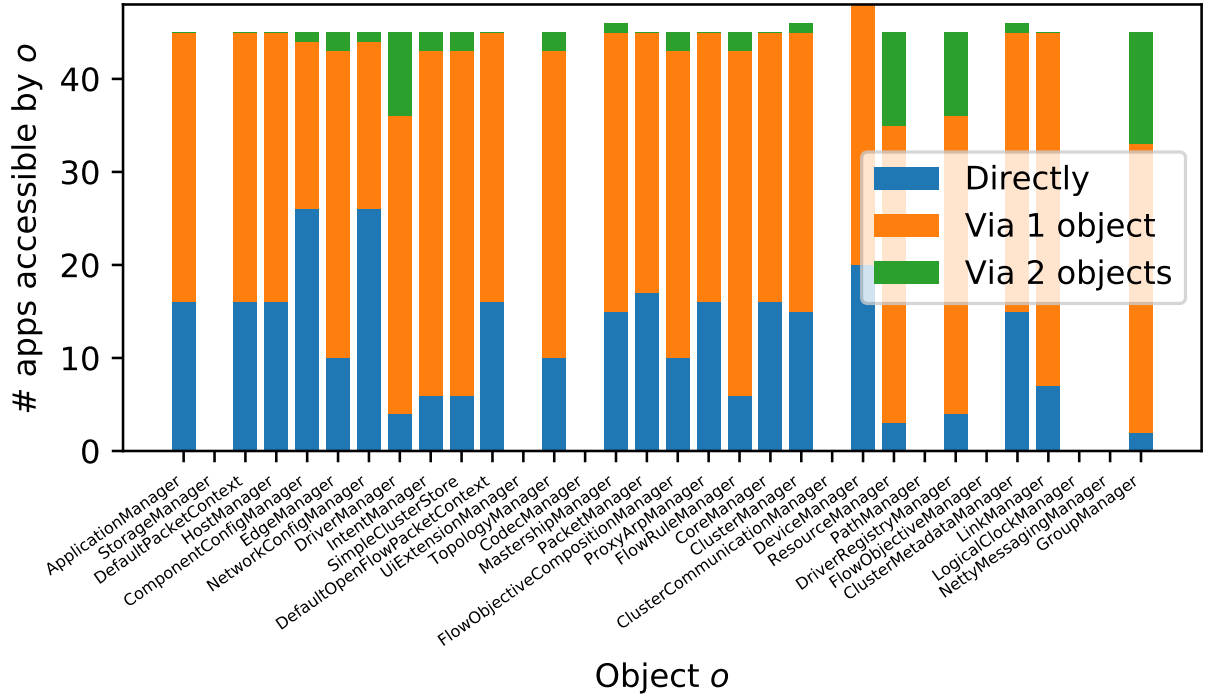


Figure 4.4: Object to app accessibility (via shortest paths) in \mathcal{G}_{ONOS} with 63 apps. Paths begin at a given object o .

test from consideration, since it is used for testing ONOS functionality.

4.5.2 CAP Gadgets in Security-Mode ONOS

We further refine the results from Figures 4.3 and 4.4 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.

4.5.2.1 Methodology

We used JavaParser [167] 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 [168] for C/C++ or ast [169] for Python exist for controllers and apps in other languages.

4.5.2.2 Results

Table 4.1 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*.

Table 4.1: 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>i.e.</i> , 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	sdnnp	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>i.e.</i> , 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

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.

4.5.3 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 B.1.1 for configuration details.)

4.5.3.1 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-service (DoS). Our approach is similar to the attacks proposed by Dhawan *et al.* [93] and Lee and Shin [113] 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.2 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`.

4.5.3.2 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.

4.5.4 Remarks

We were able to systematically detect CAP gadgets (as described in Section 4.5.2) 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 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.

4.6 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 [163] and on previous IFC policies that use data provenance [161, 170]. 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 $A = \{a_1, a_2, \dots, a_x\}$.
- A set of integrity tags, denoted by $T = \{\tau_1, \tau_2, \dots, \tau_t\}$.
- 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 $Ch \in \{\text{READS}, \text{WRITES}\}$.
- A response to perform when information flow is violated, denoted by $Re \in \{\text{BLOCK}, \text{WARN}, \text{NONE}\}$.

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 $A_N = \{a_1, a_2, \dots, a_n\}$ 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.

⁸For reasons explained in Section 4.7.1, we count switches as “apps.”

4.7 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.

4.7.1 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 [171, 172, 173, 174, 175], operating systems [176, 177, 178, 179], mobile phones [180, 181], and browsers [182, 183]. Provenance can be used not just for IFC but also for information tracing, accountability, transparency, and compliance [184, 185].

We use the W3C PROV data model [184, 186], 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. Entities 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 [179, 187, 188]. (See Table B.2 in Appendix B.3 for a visual representation of provenance objects and relations.)

4.7.1.1 Entities

We define *entities* as the objects from Section 4.6, 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 4.5, 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.

4.7.1.2 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.

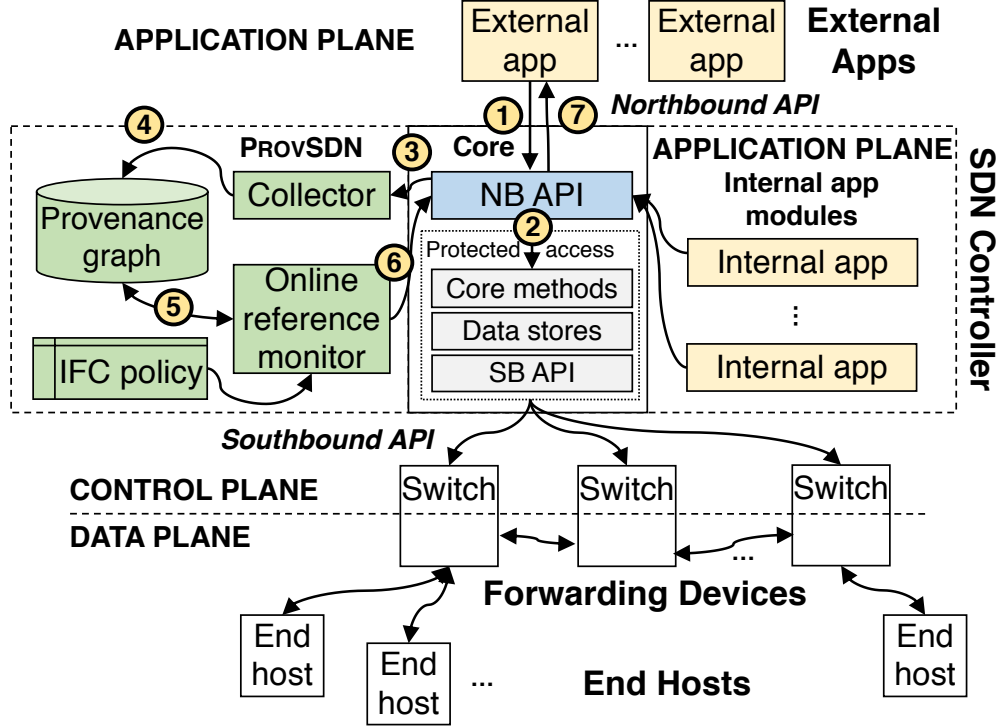


Figure 4.5: PROVSDN architecture showing an app calling the NB API. **1:** An app makes a NB API request. **2:** The NB API tentatively retrieves or inserts data related to the request. **3:** The collector processes the call information. **4:** The collector writes the provenance data to the provenance 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 .

4.7.1.3 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 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.

4.7.2 System Components

Figure 4.5 shows the PROVSDN architecture. We assume that the provenance components are trusted and adequately secured.

⁹Internal 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.

4.7.2.1 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.

4.7.2.2 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 removed¹⁰ and the action is disallowed. Otherwise, the relationship is permanently added to the provenance graph.

4.7.2.3 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.

4.7.3 Implementation

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

¹⁰To 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.

4.7.3.1 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 [189] 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 4.5). 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.

4.7.3.2 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 B.4 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.

4.7.3.3 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 [190] graph structure for optimized graph search (*i.e.*, path existence) performance.

4.7.4 Attack Evaluation

We evaluated PROVSDN's IFC capabilities using the attack described in Section 4.5.3. We prevent information flow from the triggering app (trigger) to the reactive forwarding app (fwd) by assigning

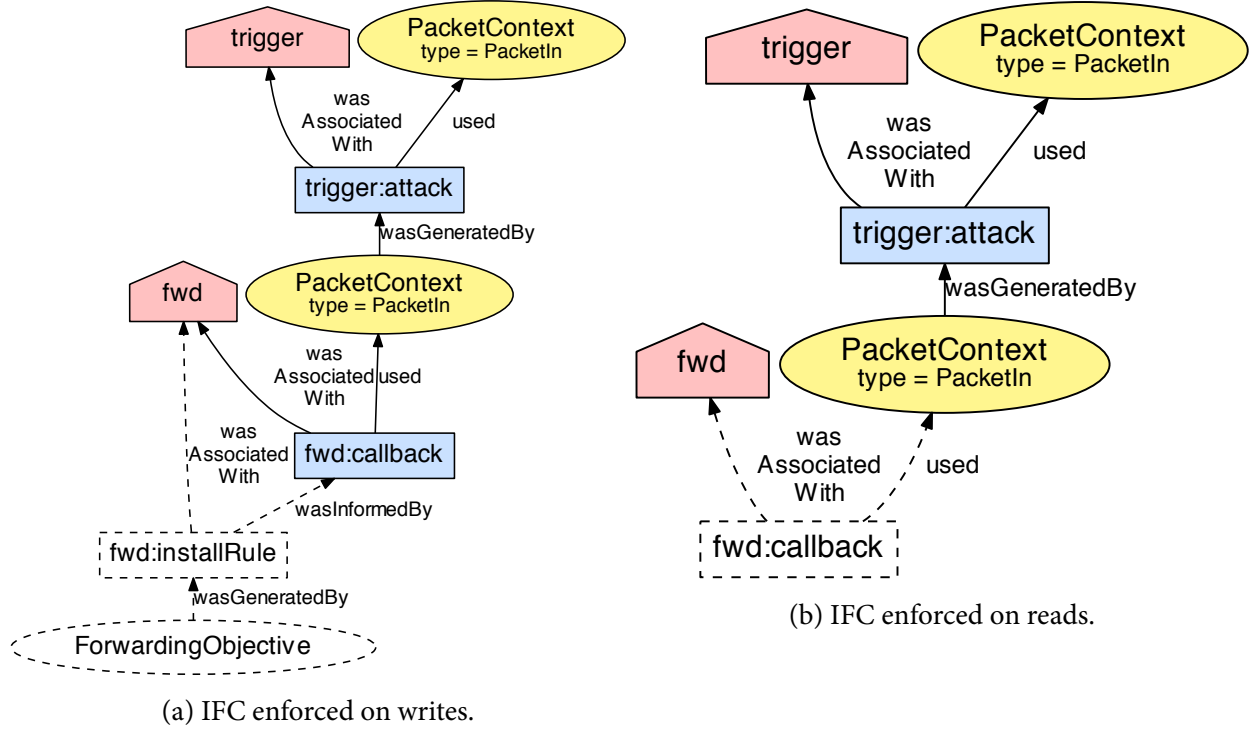


Figure 4.6: Provenance graphs generated from example CAP attack described in Section 4.5. Dashed nodes and edges represent attempted actions blocked (but recorded) by PROVSDN.

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 $Re = \text{BLOCK}$. 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 4.6 shows parts of the provenance graphs generated from the information flow attempts. If $Ch = \text{WRITES}$, IFC is enforced during write attempts, resulting in the process shown in Figure 4.6a. Similarly, if $Ch = \text{READS}$, IFC is enforced during read attempts, resulting in the process shown in Figure 4.6b. 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 4.5.3 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 4.6a). The graphical representation lets the practitioner start at the ForwardingObjective entity and trace back what data were used in generating the flow rule

Table 4.2: PROVSDN Micro-Benchmark Latencies.

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

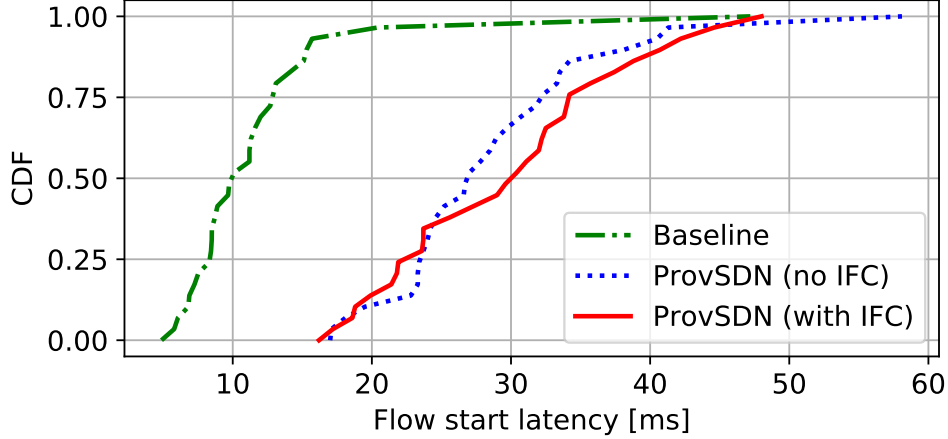


Figure 4.7: Flow start latency macrobenchmarks.

to see that trigger modified the original PacketContext entity. To prevent future occurrences, the practitioner installs the IFC policy described earlier.

4.7.5 Performance Evaluation

We evaluated PROVSDN’s performance in an emulated environment running Open vSwitch 2.7.0 [40] 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.

4.7.5.1 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 4.7 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.

4.7.5.2 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 4.5). From Table 4.2, 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 μ s 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.

4.8 Discussion

Extent to Which Controllers Are Affected by CAP OpenDaylight [30] 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" [191].

Floodlight [28] 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," [192] which would make CAP attacks trivial. SE-Floodlight [15] 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 [29], 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 [17], for instance, includes sub-method permissions such as allowing or denying flow entries based on IP source and destination prefixes. (See Appendix B.1 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) [193] and history-based access control (HBAC) [194] propose IFC for Java-based systems. Jif [195] 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 [161]. 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 [196] 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 [197], like PROVSDN, uses a low-watermark floating label policy [162] for Android to prevent permission re-delegation. Quire [181] 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 [198] 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.* [199] implemented and formally verified an IFC extension to the Chromium Web browser that uses lightweight 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 [197, 200] 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 B.4.

4.9 Related Work

SDN Controller Security Wen *et al.* [17] 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 4.2, 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 [19] extends the ONOS controller to include API method-level RBAC enforcement. Rosemary [14] isolates applications by running each application as an individual process. SE-Floodlight [15] hardens the control plane by enforcing hierarchical RBAC policies and logging events through an auditing subsystem. These systems neither explicitly track information flow necessary for detecting CAP attacks nor enforce IFC policies in real time as can be done with PROVSDN. FRESCO [5] 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 [115, 116, 117, 119, 201]. 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 scope.

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 [156]. Several efforts [113, 155] 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 [113, 155] 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.* [112].

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 [96] and NetPlumber [97], 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 [105] verifies that an application cannot install flow rules that violate a set of constraints, but does not consider controller–application interactions. DELTA [112], ATTAIN [81], and BEADS [111] 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 [141] 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.* [122] 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.* [132] leverage meta-provenance to facilitate automated repair of a network. Bates *et al.* [3] demonstrate a way to improve a previous approach [144] 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.* [144] consider the task of identifying malicious nodes in a distributed system. Chen *et al.* [202] 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 [131].

4.10 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.

CHAPTER 5

CONTROL PLANE EVENT-BASED VULNERABILITIES

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

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

5.1 Introduction

Software-defined networking (SDN) has experienced a rapid rise in adoption within data center providers, telecommunication providers, and other enterprises because of its programmable and extensible control plane [1]. SDN claims to decouple the network's decision-making about forwarding (*i.e.*, the *control plane*) from the traffic being forwarded (*i.e.*, the *data plane*) so as to allow centralized oversight through an *SDN controller* and *network applications* (or *apps*) in the enforcement of consistent (security) policies.

All popular modern SDN controllers, including ONOS [25], OpenDaylight [30], Hewlett Packard Enterprise's VAN SDN Controller [152], and Floodlight [28], operate as reactive *event-driven architectures* that, based on data plane activities, use asynchronous event dispatchers, event listeners, and

controller API calls to pass information among controller and app components.¹ Each app's event listeners subscribe to a subset of the possible universe of events. Based on the event, an app may call API services (e.g., a request to insert a new flow rule) or generate new events (e.g., a notification that a new host has been seen in the data plane).

SDN's programmability significantly alters the control plane's attack surface. The claim of control and data plane decoupling belies a subtle and serious challenge: control plane decisions are often made as a result of information collected from an untrustworthy data plane. Prior attacks [22, 55, 93] have demonstrated specific examples of what we generalize as the class of *cross-plane attacks*, which allow attackers to influence control plane decision-making without attacking the controller or apps directly [80]. For instance, a clever attacker who controls a data plane host can emit packets that are acted upon by controller and app components, which can result in malicious privilege escalation or malicious control over flow rule behaviors by a host.

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

The vulnerabilities that result from complex event and app interactions are challenging to detect automatically because such vulnerabilities are a class of *logic* (or *semantic*) *bugs* that require local and global semantic understanding about events and their use. Logic bugs are of interest to attackers because such bugs are difficult to identify during software development and can persist for years before disclosure [203]; existing tools often focus on bugs related to language grammar or resource use only (e.g., FindBugs [204], PMD [205], and Coverity [206]) or require developers to annotate code (e.g., KINT [207]), rendering such tools difficult to use in practice [208, 209].

In the absence of developer annotations that specify intended app behavior, the vulnerability search space can become large [105, 110, 112] [208]. However, by focusing on a narrower scope of event-related vulnerabilities that involve *missing* or *unhandled events*, we can tractably enumerate those conditions and investigate them. Uncovering such vulnerabilities requires understanding of how events are used within SDN components, how events are passed between SDN components,

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

and how events’ actions propagate within the control plane to have data plane effects. Given the event-driven nature of modern SDN architectures, our insight is that *event-related bugs that result from unhandled events are of high interest in SDN security evaluation*, particularly if cross-plane attacks can be used to trigger such vulnerabilities that ultimately lead to data plane consequences (e.g., flow rule installation).

Although tools have been developed to perform vulnerability discovery in SDNs with fuzz testing [111, 112], concurrency detection [109], and code analysis [113, 114], we are not aware of any tools that are designed specifically to aid developers and practitioners in the understanding of global event use and in the identification of unhandled event vulnerabilities at design and testing time. Forensic SDN tools [51, 129] provide causal explanations of past executions but do not identify vulnerabilities ahead of time.

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

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

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

Using the event flow graph, EVENTSCOPE then validates whether potentially missing events can cause data plane effects in the presence or absence of other apps. Given an app with such a candidate vulnerability, EVENTSCOPE identifies other apps that handle that app’s missing event and also have data plane effects to create a *context* for that vulnerability. Next, EVENTSCOPE represents these code

executions as event flow graph paths to determine whether they have data plane effects. Finally, EVENTSCOPE generates a list of vulnerabilities for analysis by developers and practitioners.

We use the open-source, Java-based ONOS SDN controller [25] as a representative case study. ONOS is used in production settings by telecommunications providers, and its codebase underlies proprietary SDN controllers developed by Ciena, Samsung, and Huawei [159]. ONOS’s extensive event-centered design makes the controller an ideal candidate for study. We analyzed how ONOS’s core service and app components use events, discovering that many events are not handled even when components subscribe to those events. Although we focus on ONOS as a case study, we note that all modern SDN controllers use a similar event-based architecture; thus, EVENTSCOPE’s methodology is broadly applicable to all such controllers.

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

Contributions Our main contributions are:

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

Organization This chapter is organized as follows. In Section 5.2, we explain the challenges to event-driven SDN architectures and our mitigations. In Section 5.3, we provide an overview of

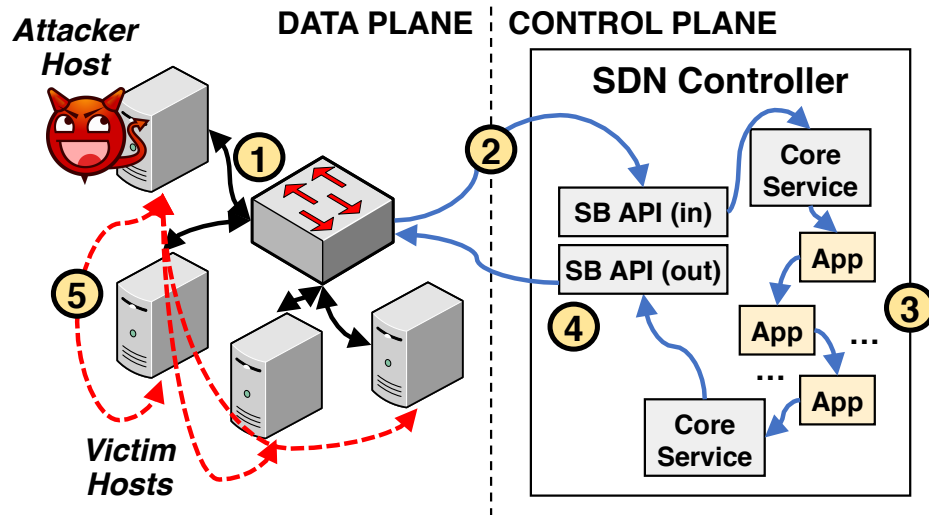


Figure 5.1: Cross-plane attack example. Black arrows denote data plane connections, blue arrows denote control plane control flow, and red arrows denote intended effect (e.g., increased data plane access). 1: An attacker emits data plane packets. 2: The controller’s southbound API receives packets. 3: The controller’s components use the data plane input to make a data plane decision. 4: The controller emits new packets or flow rules into the data plane. 5: The attacker uses the new packets or flow rules as a step to actuate an attack.

EVENTSCOPE. In Section 5.4, we propose an event use methodology to identify candidate vulnerabilities. In Section 5.5, we analyze intra-component and inter-component uses that are abstracted into event flow graph structures to validate vulnerabilities. In Section 5.6, we discuss implementation details. In Section 5.7, we demonstrate the effectiveness of our approach by describing vulnerabilities found in ONOS. In Section 5.8, we discuss the challenges of SDN design and vulnerability discovery. In Section 5.9, we discuss related work. In Section 5.10, we conclude.

5.2 Challenges

We outline the challenges and mitigation approaches for SDN security that are related to adversarial data plane input, event-driven apps, and event flow interactions. Although we use the ONOS SDN controller as a running example, we note that other SDN controllers (e.g., Floodlight [28]) share similar event-driven features.

5.2.1 Malicious Data Plane Input

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

to influence control plane behavior even if the controller and app infrastructures are assumed to be hardened. *Cross-plane attacks*, such as topology poisoning [22, 55, 93], impact control plane operations by causing denial-of-service or connectivity-based attacks. Figure 5.1 shows a representative example of a cross-plane attack that uses malicious data plane input to produce an unintended data plane effect.

Attackers can infer whether the network is non-SDN or SDN and which controller is being used in an SDN setting [64, 65]. Defenses to date, such as control plane causality tracking [51, 129], trusted data plane identities [58], and timing-based link fabrication prevention [55], are useful in preventing specific classes of attacks but are not designed for vulnerability discovery because they track specific execution traces as they occur rather than all possible execution traces prior to runtime. Current SDN vulnerability tools, such as BEADS [111] and DELTA [112], rely on fuzzing techniques that do not easily capture complex event-based vulnerabilities.

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

Our mitigation approach EVENTSCOPE analyzes how malicious data plane input and cross-plane attacks can have cascading effects throughout controller components and apps as a result of unhandled event types (Section 5.5). We demonstrate how that analysis allows us to identify ONOS app vulnerabilities (Section 5.7).

5.2.2 Event-Driven Apps

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

Network verification approaches [96, 105] require formal property specifications or do not scale beyond trivial controllers. CONGUARD [109] and DELTA [112] offer models for reasoning about the ordering of OpenFlow events, but such events are only one part in a complex, event-driven, network operating system that must consider additional (and often more sophisticated) network abstractions.

Our mitigation approach EVENTSCOPE uses a clustering approach to infer the intended application behavior based on the insight that apps that perform similar functionality are interested in similar kinds of events and event types (Section 5.4). EVENTSCOPE identifies cases in which a given app’s event types are absent with respect to similar apps and evaluates whether these absences create vulnerabilities (Section 5.5.2).

5.2.3 Event Flow Interactions

As apps can originate from different parties [51], assessment of system-wide “correct” behavior is complex when components closely collaborate and form event-driven dependencies. The event-driven SDN architecture allows flexible and composable development, with events helping to provide convenient abstractions and allowing components to subscribe to asynchronous activities of interest. Prior work [115, 116, 117, 119, 201] has approached controller design by providing formally specified runtime languages and safe-by-construction controllers, but such approaches do not offer the extensibility of the operating-system-like controllers used in practice in production.

Understanding how event-driven components in an SDN interact is challenging because events have both control flow and data flow elements. Events represent *control flow* because they are processed by event listener methods that may call additional methods depending on the event information, and they represent *data flow* because they carry data describing the event (*e.g.*, a host event contains that host’s details). Although control flow and data flow can be modeled together in program dependence graphs [210] or code property graphs [211], analysis is often limited to single procedures because too many details prevent the analysis from scaling to complex, inter-procedural event-driven systems. Further, events can be used to influence what code paths are or are not taken and to trigger additional events.

Our mitigation approach EVENTSCOPE uses the event flow graph to model the key features of an event-driven SDN system while abstracting away unnecessary control flow details (Section 5.5.1). The event flow graph shows how triggered events have consequences elsewhere, particularly when malicious data plane inputs later influence data plane changes.

5.3 EVENTSCOPE Overview

We designed EVENTSCOPE to identify cross-plane event-based vulnerabilities in three phases, as illustrated in Figure 5.2.

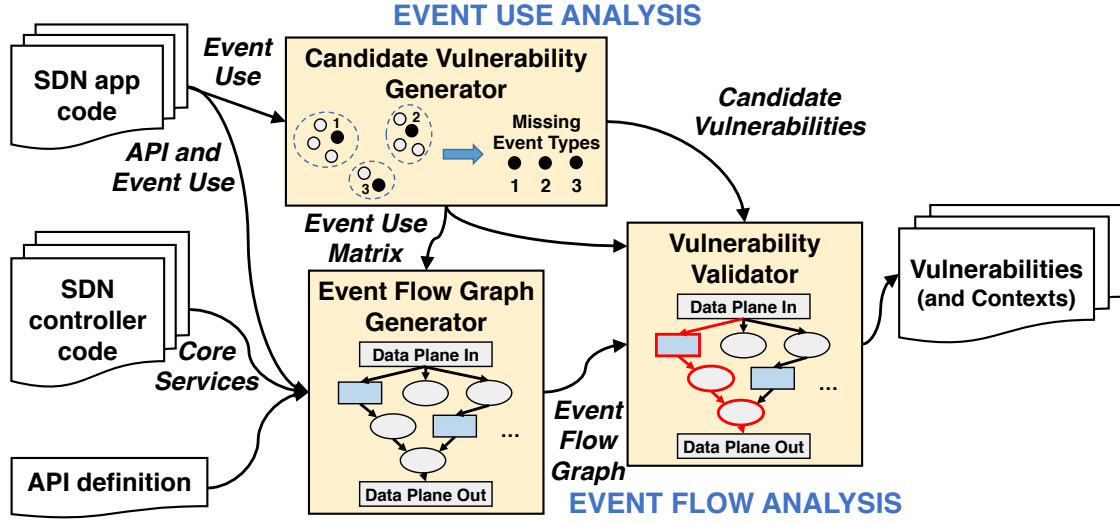


Figure 5.2: EVENTSCOPE architecture overview.

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

The second phase, the *event flow graph generator*, takes the apps' code, the controller's code, and a definition of controller API calls as inputs and constructs an event flow graph that records how events propagate and influence the system. This includes event propagation within the controller as well as within apps and combinations of apps.

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

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

- The **candidate vulnerability generator** (Section 5.4) generates a list of possible vulnerabilities resulting from unhandled events based on apps' event use in comparison to that of similar apps.
- The **event flow graph generator** (Section 5.5.1) analyzes the use of events between components to construct a concise representation of how events are passed and how they affect data plane operations.

- The **vulnerability validator** (Section 5.5.2) filters and validates the possible missing-event-handling vulnerabilities from the first component by using the event flow graph to determine whether the missing event has had data plane impacts, either in isolation or in combination with other apps.

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

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

5.4 Event Use Analysis

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

5.4.1 Event Use Methodology

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

5.4.1.1 Algorithm

We describe EVENTSCOPE’s approach, shown in Algorithm 5.1. We assume a set of apps that contain event listeners, A ; a set of event kinds, E_K (*e.g.*, HostEvent in ONOS); a set of event types, E_T (*e.g.*, HOST_ADDED in ONOS) that relate to the functional nature of event kinds in E_K ; and a threshold, τ , used to determine the number of app clusters. For intermediate data structures, we generate an event use matrix, M , that shows how apps use event types; a distance matrix, D , that

Algorithm 5.1 Candidate Vulnerability Generation

Input: Apps A , event kinds E_K , event types E_T , threshold τ
Output: List of candidate vulnerabilities V_C , event use matrix M
Initialize: $M[i][j] \leftarrow \text{false}; \forall i \in A, \forall j \in E_T$ ▷ Event use matrix $M_{A \times E_T}$
 $D[i][j] \leftarrow 0; \forall i \in A, \forall j \in A$ ▷ Distance matrix $D_{A \times A}$
 $\mathcal{V} \leftarrow A \cup E_T, \mathcal{E} \leftarrow \emptyset, \mathcal{G}_S \leftarrow (\mathcal{V}, \mathcal{E})$ ▷ SimRank graph \mathcal{G}_S
 $V_C \leftarrow \emptyset$ ▷ Candidate vulnerability list V_C
1: **for each** $a \in A$ **do**
2: $T \leftarrow \text{getHandledEventTypes}(a)$
3: **for each** $t \in T$ **do**
4: $M[a][t] \leftarrow \text{true}$
5: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(a, t), (t, a)\}$
6: **end for**
7: **end for**
8: $S \leftarrow \text{SimRank}(\mathcal{G}_S, A)$ ▷ Similarity matrix $S_{A \times A}$
9: **for each** $i \in S$ **do**
10: **for each** $j \in S[i]$ **do**
11: $D[i][j] \leftarrow 1 - S[i][j]$ ▷ Distance = 1 - Similarity
12: **end for**
13: **end for**
14: $C \leftarrow \text{hierarchicalCluster}(D, \tau)$ ▷ Set of app clusters C
15: **for each** $c \in C$ **do**
16: $u \leftarrow \emptyset$ ▷ Union of event types within cluster c
17: **for each** $a \in C$ **do**
18: $u \leftarrow u \cup M[a]$
19: **end for**
20: **for each** $a \in C$ **do**
21: $d \leftarrow u \setminus M[a]$ ▷ Set difference d of cluster and app
22: **for each** $t \in d$ **do**
23: $k \leftarrow \text{getEventKind}(t, E_K, E_T)$
24: **if** k is handled by a **then**
25: $V_C.\text{append}((a, t))$
26: **end if**
27: **end for**
28: **end for**
29: **end for**

represents the “distances” between apps in terms of how they are related; and a bipartite directed graph, \mathcal{G}_S , that represents the relations between apps and event types.

The algorithm determines the event types that each app uses (lines 1–5). It does so using static analysis through the generation of a control flow graph (CFG) of the relevant event listener method. If a given event type is handled (line 2), it is marked in the event use matrix, M , (line 4) and in the bipartite graph, \mathcal{G}_S (line 5). The algorithm then computes the SimRank similarity metric across \mathcal{G}_S and reduces it to vertices of interest, or $A \subset \mathcal{V}$, to produce the similarity matrix, S (line 5). It then

takes the inverse of the similarity metric to compute the distance metric (lines 7–9), and uses it to compute app clusters by using a complete-linkage² (*i.e.*, maximum linkage) hierarchical clustering algorithm (line 10).

After the apps are partitioned into clusters, the algorithm inspects each app relative to its own cluster (lines 11–20). For each cluster, it generates a union of event types handled by that cluster’s apps (lines 12–14). For a given app, it computes what event types are not handled by that app’s event listener with respect to the cluster’s union (line 16). In some cases, the event type will be related to an event kind that the app does not handle at all, and we do not consider such scenarios to represent candidate vulnerabilities. When the event type’s kind *is* handled by the app (line 19), the algorithm marks the event kind as a candidate vulnerability (line 20).

5.4.1.2 Design decisions

Initially, we applied the Levenshtein distance as our distance metric by treating each row of M as a bit vector, based on prior work on SDN app API use similarity [114]. However, we found that the Levenshtein distance did not capture the structural similarities among apps, event kinds, and event types. Instead, we opted for the SimRank metric, which expresses the idea that “two objects are similar if they are related to similar objects” [213]. SimRank fits more naturally with our problem of expressing the similarity of two apps that have relations to similar event types.

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

5.4.1.3 Interpretation

Because apps do not provide well-defined semantics about their correct operation, we do not have ground truth about what event types each app *should* handle. As a result, we chose to focus on

²Alternatives include single-linkage and average-linkage clustering. We chose complete-linkage clustering because it 1) maximizes the distance between two elements of different clusters and 2) avoids the problem of grouping dissimilar elements that single-linkage clustering would entail [212].

instances of missing event handling, which we can identify based on knowledge about the complete set of events. Unfortunately, such instances do not tell us the extent to which such missing events are intentional or the extent to which missing events’ exploitation can cause unexpected behavior. While any instance is arguably a concern, we wanted to focus our effort on the instances *most likely* to be vulnerabilities. As a result, we chose to cluster apps in order to identify the missing event handling that stands out as the most “unusual,” with the parameter τ approximating the unusualness of missing event handling.

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

5.4.2 Event Use Results

We evaluated EVENTSCOPE’s event use analysis using ONOS v1.14.0 [26]. In addition to ONOS’s core services, the ONOS codebase includes third-party apps written by independent developers. We explain each part of the methodology as applicable to ONOS and its apps.

5.4.2.1 ONOS’s event system

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

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

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

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

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

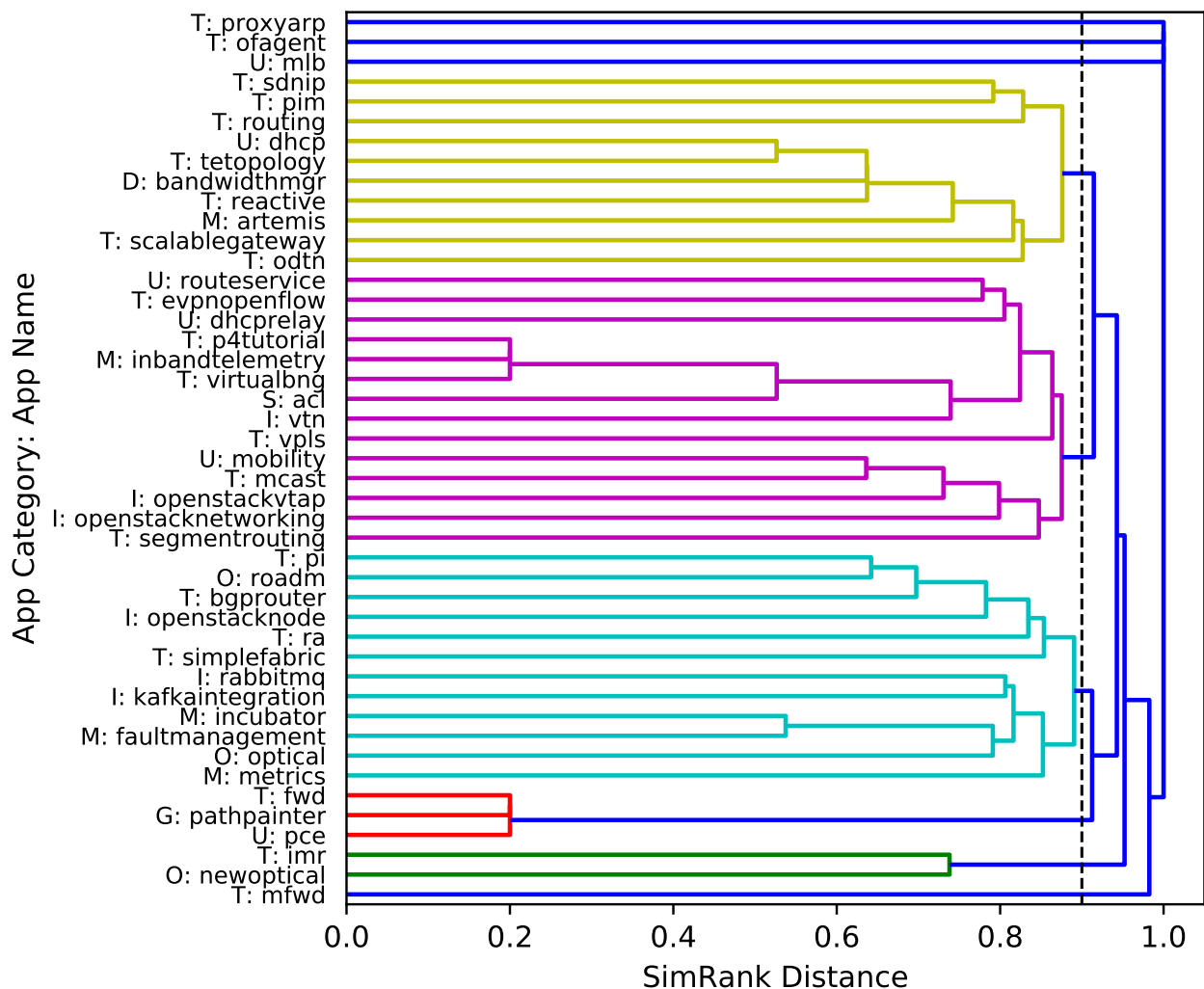


Figure 5.4: Dendrogram representation of ONOS network event type similarity among apps, based on the SimRank distance metric. The dashed vertical line represents a threshold $\tau = 0.90$ with a partitioning of 9 clusters.

5.4.2.2 ONOS unhandled event types

Figure 5.3 shows EVENTSCOPE’s generated event use matrix M of the 45 apps included with the ONOS codebase. Each ONOS app includes a self-defined category, and categories are grouped by horizontal dividers. Each event kind is grouped by vertical dividers. Figure 5.4 shows the dendrogram of the resulting app clusters, based on SimRank distance and complete-linkage clustering.

We empirically chose a threshold ($\tau = 0.90$) that yielded a number of clusters (*i.e.*, 9) similar to the number of categories of ONOS apps (*i.e.*, 8) based on the assumption that there exist at least as many categories as there are functional differences among apps. We found that that threshold worked well in the rest of our evaluation. (See Appendix C.3 for an evaluation of τ on detection rates.) We found that setting the threshold too low (*i.e.*, more clusters) created more singleton app clusters,

which should be avoided because each cluster’s union of event types becomes the event types the app handles. However, setting the threshold too high (*i.e.*, fewer clusters) clustered apps with too few functional similarities. Based on that threshold, we generated 116 candidate vulnerabilities, which were used as input into the next stage of EVENTSCOPE (Section 5.5).

5.5 Event Flow Analysis

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

5.5.1 Event Flow Graph Generation

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

5.5.1.1 Definitions

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

Formally, an *event flow graph*, denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a directed, multi-edged graph that models the abstractions for inter-procedural and inter-component control and data flows in the SDN control plane. Event flow graphs summarize the necessary control and data flows among components

⁵In traditional static analysis, a program has a well-defined entry point: the `main()` function. However, since SDN is event-driven, no `main()` function exists [129]. To correct for the lack of a `main()` function and to account for the event-driven architecture, we use each component as an entry point.

⁶Lu *et al.* [214] define that as “splitting” in the component hijacking problem.

needed for event flow analysis. Vertices, denoted by \mathcal{V} , consist of one of the following types: event listeners (represented as entry point methods), API services (represented as an API interface method or its implemented concrete method), and representations of data plane input (DPI_{in}) and data plane output (DPO_{out}). Edges, denoted by \mathcal{E} , are labeled and consist of one of the following types: API read calls (API_READ), API write calls (API_WRITE), data plane inputs to methods (DP_IN), methods' output to the data plane (DP_OUT), or passing of an event type (*e.g.*, HOST_ADDED event type of the HostEvent event kind).

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

5.5.1.2 Methodology

EVENTSCOPE's approach is shown in Algorithm 5.2. It initializes the event flow graph's vertices to be the set of event listeners and representations for data plane inputs and outputs. It begins with the set of event listeners as the components of entry points to check (line 1). For each entry point, it generates a call graph (line 5). Within the call graph, it checks whether calls relate to an API read (lines 7–10), to an API write (lines 11–14), or to the event dispatcher to generate new events (lines 15–16). It links the event dispatchers and event listeners together in the event flow graph by using the event use matrix, M , generated in the prior step (Section 5.4); each event type that is handled by a particular listener is represented as its own edge, so multi-edges are possible (line 18). Finally, it identifies core service components that take in data plane input or generate data plane output, and links those to the data plane input and output vertices (line 19).

5.5.1.3 Results

To show how an event flow graph abstracts useful information for understanding SDN architecture events, we consider the partial event flow graph from ONOS shown in Figure 5.5. It shows the forwarding app (fwd) packet processor component as an entry point. (For event flow graphs that include event dispatch edges, see Figures 5.7 and 5.8 in Section 5.7 and Figure C.2 in Appendix C.2.) General static analysis tools produce control flow graphs (CFGs) for each procedure or method, as well as a call graph (CG) for inter-procedural analysis; however, static analysis tools face challenges

Algorithm 5.2 Event Flow Graph Generation

Input: API read methods A_r , API write methods A_w , data plane input methods D_i , data plane output methods D_o , event listener methods E_l , event kinds E_K , event types E_T , event use matrix M

Output: Event flow graph \mathcal{G}

Initialize: $\mathcal{V} \leftarrow E_l \cup \{\text{DPIn}, \text{DPOut}\}$, $\mathcal{E} \leftarrow \emptyset$

$S \leftarrow E_l$ ▷ Stack S of entry points (*i.e.*, components) left to check

$C \leftarrow \emptyset$ ▷ Checked components C

$E_d \leftarrow \emptyset$ ▷ Components that dispatch events E_d

```
1: while  $S$  is not empty do
2:    $e \leftarrow S.\text{pop}()$  ▷ Entry point method  $e$ 
3:   if  $e \in C$  then
4:     continue ▷ Skip entry point if already processed
5:   end if
6:    $(c_v, c_e) \leftarrow \text{generateCG}(e)$  ▷ Call graph vertices  $c_v$  and edges  $c_e$ 
7:   for each  $c \in c_v$  do
8:     if  $c \in A_r$  then
9:        $\mathcal{V} \leftarrow \mathcal{V} \cup \{c\}$ 
10:       $\mathcal{E} \leftarrow \mathcal{E} \cup \{(c, e)\}$  ▷ Labeled edge API_READ
11:       $S.\text{push}(c)$ 
12:    else if  $c \in A_w$  then
13:       $\mathcal{V} \leftarrow \mathcal{V} \cup \{c\}$ 
14:       $\mathcal{E} \leftarrow \mathcal{E} \cup \{(e, c)\}$  ▷ Labeled edge API_WRITE
15:       $S.\text{push}(c)$ 
16:    else if  $c$  is the event dispatch method then
17:       $E_d \leftarrow E_d \cup \{(c)\}$ 
18:    end if
19:  end for
20:   $C \leftarrow C \cup e$ 
21: end while
22:  $\mathcal{E} \leftarrow \text{linkListenersDispatchers}(\mathcal{E}, E_d, E_l, E_K, E_T, M)$  ▷ Labeled edges of particular event type  $t \in E_T$ 
23:  $\mathcal{E} \leftarrow \text{linkDataPlane}(\mathcal{E}, D_i, D_o)$  ▷ Labeled edges DP_IN or DP_OUT
24:  $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E})$ 
```

regarding the understanding of API behavior and the semantics of a given program's domain [209]. While both CFGs and a CG are necessary for control or data flow analyses, neither type of graph represents the SDN domain's semantics of events or API behavior at the right level of abstraction.

We generated an ONOS event flow graph whose components include core services, providers,⁷ and 45 apps. The ONOS event flow graph's nodes consists of representations of 143 event listeners, 25 packet processors, 81 API call methods of core services, 1 data plane input node, and 1 data plane output node. The ONOS event flow graph's edges consist of representations of 396 API calls, 352 event dispatches, and 21 data plane interactions. Appendix C.2 shows a partial representation of that

⁷In ONOS, a *provider* interacts with core services and network protocol implementations [32]. We consider provider services to be core services.

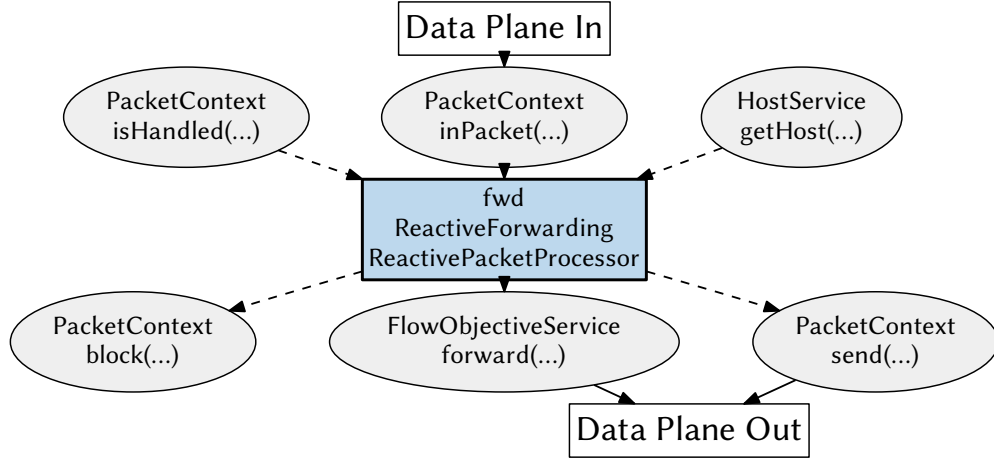


Figure 5.5: Event flow graph of fwd’s packet processor. Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, and dashed edges represent API calls.

event flow graph based on 5 sample apps and the core services that they use.

Because ONOS does not specify a precise set of API calls that comprise the northbound API [51], we used the public method signatures of the **Service* and **Provider* classes, along with those methods’ return values, to determine API read and write calls, resulting in 123 API read call methods, 87 API write call methods, 1 method directly related to data plane input, and 44 methods directly related to data plane output and effects. We identified event dispatching based on direct calls to the event dispatcher for local events (*e.g.*, `post()`) or indirect calls to a store delegate⁸ for distributed events.

5.5.2 Vulnerability Validation

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

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

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

5.5.2.1 Context

Event handling vulnerabilities do not occur in isolation, but as part of a complex interaction web involving many other event handlers and apps. We need to consider that context when discussing a given vulnerability. We borrow from Livshits and Lam [215] the intuition that exploitable vulnerabilities can occur as a result of a multi-stage exploit via an initial data injection and a subsequent app manipulation. As a result, we define the *present context* as the set of other apps that 1) handle the vulnerability’s missing event type in the absence of the vulnerable app’s event handler’s handling of it, 2) are affected by data plane input, and 3) have data plane effects. We define *absent context* as the set of other apps that, like the app in question, do not handle the vulnerability’s missing event type but can be affected by data plane input and have data plane effects.

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

We note that exploit generation is nontrivial and that automatic exploit generation [216] is an ongoing research area. EVENTSCOPE’s output includes “valid” vulnerabilities and contexts that EVENTSCOPE believes to be reachable from the data plane and to have data plane impacts. While EVENTSCOPE’s validation provides strong soundness properties, static analysis is necessarily imprecise; manual verification is still recommended. EVENTSCOPE provides precisely the details that need to be included in a bug report. However, the tool neither provides a guarantee that a bug exists nor automatically submits bug reports. For the vulnerabilities EVENTSCOPE found, we manually examined the source code to confirm that the vulnerabilities existed.

5.5.2.2 Methodology

EVENTSCOPE’s approach for vulnerability validation is shown in Algorithm 5.3. It uses the event flow graph, candidate vulnerabilities, and the event use matrix as inputs. Each candidate vulnerability is represented as a tuple of the app and unhandled event type (line 1). For each event type, EVENTSCOPE gets the app’s event listeners (line 2). It performs path connectivity queries over the event flow graph. If at least one path does not exist that starts from the data plane, goes through one of the app’s event handlers, and ends in the data plane, then the algorithm does not consider a vulnerability to be relevant, either because the event listener is not affected by data plane input or because the resulting path does not have data plane effects (lines 3–4).

The algorithm initializes the present and absent context sets to be empty (line 5). It inspects all of the other apps in the event flow graph to build the context (line 6). If another app’s event listener is affected by data plane input and has data plane effects (line 8), it checks whether the missing event

Algorithm 5.3 Vulnerability Validation

Input: Event flow graph \mathcal{G} , list of candidate vulnerabilities V_C , event use matrix M , apps A
Output: List of vulnerabilities and contexts V
Initialize: $V \leftarrow \emptyset$ ▷ Vulnerabilities and contexts list V

```
1: for each  $(a, t) \in V_C$  do ▷ App  $a \in A$  and event type  $t \in E_T$   
2:    $E_l \leftarrow \text{getEventListeners}(a, \mathcal{G})$  ▷  $E_l \subset \mathcal{G}$ 's vertices  
3:   if  $\neg(\text{pathExists}(\text{DPIIn} \rightarrow e \in E_l \rightarrow \text{DPOut}, \mathcal{G}))$  then  
4:     continue  
5:   end if  
6:    $c_+ \leftarrow \emptyset, c_- \leftarrow \emptyset$  ▷ Present context set  $c_+$ , absent context set  $c_-$   
7:   for each  $a_i \in A \setminus \{a\}$  do ▷ All apps except  $a$   
8:      $E_{l_i} \leftarrow \text{getEventListeners}(a_i, \mathcal{G})$  ▷  $E_{l_i} \subset \mathcal{G}$ 's vertices  
9:     if  $\text{pathExists}(\text{DPIIn} \rightarrow e \in E_{l_i} \rightarrow \text{DPOut}, \mathcal{G})$  then  
10:      if  $t \in \text{getHandledEventTypes}(E_{l_i}, M)$  then  
11:         $c_+ \leftarrow c_+ \cup a_i$   
12:      else  
13:         $c_- \leftarrow c_- \cup a_i$   
14:      end if  
15:    end if  
16:  end for  
17:   $V.\text{append}((a, t, c_+, c_-))$   
18: end for
```

type is handled by that app (line 9) or not (line 11), and builds the context sets accordingly (lines 10 and 12). It then appends the vulnerability to the vulnerability list (line 13).

5.5.3 Performance Results

We ran EVENTSCOPE using an Intel Core i5-4590 3.30 GHz CPU with 16 GB of memory on ONOS and its associated apps. Figure 5.6 shows the cumulative distribution functions (CDFs) of the component analysis latency (Figure 5.6a) and the number of methods traversed in the call graph generation (Figure 5.6b); the latency corresponds to the computations of lines 2–17 in Algorithm 5.2, and the methods traversed correspond to line 5 in Algorithm 5.2.

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

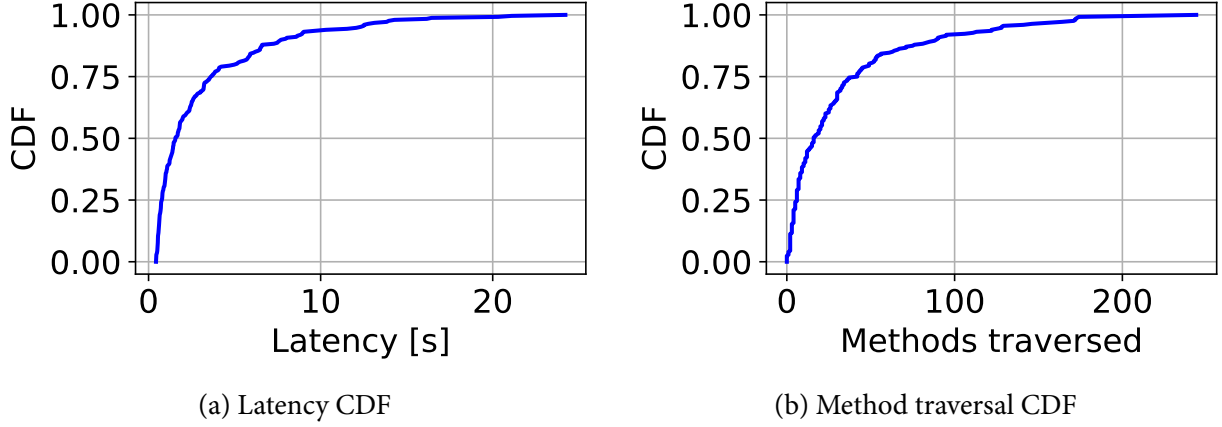


Figure 5.6: Component analysis performance results.

5.6 Implementation

We implemented EVENTSCOPE using a combination of Python and Java. In Python, we used Scikit-learn [217] to perform hierarchical clustering in the event use analysis. In Java, we used Soot [218] to generate the control flow graphs and call graphs used for event use analysis and for determining entry points. Soot creates an intermediate representation in Jimple. We also used JGraphT [190] to store in-memory representations of event flow graphs and to query path connectivity.

For connectivity queries in lines 3 and 8 in Algorithm 5.3 (*i.e.*, `pathExists()`), we used Dijkstra’s algorithm. The worst-case performance time for each `pathExists()` query can be optimized [219] to $O(2(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|))$, where $|\mathcal{E}|$ represents the number of event flow graph edges and $|\mathcal{V}|$ represents the number of event flow graph nodes. In practice, we found that the small number of apps and events did not pose a challenge for connectivity computations.

Soot operates on Java bytecode, which allows EVENTSCOPE to analyze closed-source Java-based controllers and apps. Similar program analysis tools, such as angr [220], can operate on closed-source binary executables. Using bytecode is advantageous, as we can use EVENTSCOPE to generate event flow graphs without requiring Java source code. Thus, EVENTSCOPE can be useful for practitioners as a code audit tool. Although we did not encounter any apps that used dynamic calls, such as the Java language’s reflection API, TamiFlex [221] extends Soot to perform static analysis that accounts for reflection.

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

5.7 ONOS Vulnerability Evaluation Results

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

Table 5.1 shows the 14 vulnerabilities, based on app, event kind, and unhandled event types. Table 5.1 also provides sample paths in the event flow graph. We found that all vulnerabilities involved the HostEvent event kind, which indicates that data plane input has the most effect on host information in ONOS.

EVENTSCOPE's output included 14 possible vulnerabilities. We manually investigated each vulnerability in the source code and determined that all of them could be exploited from the data plane. As a result, Table 5.1 represents EVENTSCOPE's complete output with no false positives. EVENTSCOPE's final phase essentially filters out missing event handling that cannot be reached from the data plane or trigger impacts on the data plane; as a result, the output provides strong soundness properties. As we do not have ground truth about which unhandled event types should be handled, we note that the event use analysis in Section 5.4.1 should be interpreted as a filter of the unhandled event types that are *most likely* to require attention, based on such event types' absence *vis-à-vis* a cluster of the most similar apps. As noted earlier, we chose the clustering threshold that produced a number of clusters closely matched to the number of ONOS app categories.

We describe exploits for two of the vulnerabilities below in Sections 5.7.1 and 5.7.2, and then, for the sake of space, briefly discuss the impact of the other vulnerabilities. For the exploits we created, we used a Mininet [41] SDN network. We wrote our exploit scripts in Python and used the Scapy [222] network packet library to generate data plane input.

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

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

5.7.1.1 Summary

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

Table 5.1: Event Listener Vulnerabilities Based on Event Flow Graph Analysis and Event Use Filtering ($\tau = 0.90$).

#	CVE ID	App	Unhandled type	Example event flow graph path showing potential data plane input to data plane effect
*	CVE-2018-12691	acI	HOST_UPDATED	
1	CVE-2019-11189	acI	HOST_MOVED	See Figures 5.7 and 5.8 for event flow graph examples.
2	CVE-2019-16300	acI	HOST_REMOVED	
3	CVE-2019-16298	virtualbng	HOST_MOVED	$\begin{aligned} DP_{in} &\xrightarrow{\text{inPacket()}} API_READ \xrightarrow{\text{provider.host.InternalHostProvider}} API_WRITE \xrightarrow{\text{hostDetected()}} HOST_ADDED \\ &\xrightarrow{\text{virtualbng.InternalHostListener}} API_WRITE \xrightarrow{\text{startMonitoringIp()}} DP_OUT \xrightarrow{DP_{out}} \end{aligned}$
4	CVE-2019-16298	virtualbng	HOST_REMOVED	
5	CVE-2019-16298	virtualbng	HOST_UPDATED	
6	CVE-2019-16299	mobility	HOST_ADDED	$\begin{aligned} DP_{in} &\xrightarrow{\text{inPacket()}} API_READ \xrightarrow{\text{provider.host.InternalHostProvider}} API_WRITE \xrightarrow{\text{hostDetected()}} HOST_MOVED \\ &\xrightarrow{\text{mobility.InternalHostListener}} API_WRITE \xrightarrow{\text{removeFlowRules()}} DP_OUT \xrightarrow{DP_{out}} \end{aligned}$
7	CVE-2019-16299	mobility	HOST_REMOVED	
8	CVE-2019-16299	mobility	HOST_UPDATED	
9	CVE-2019-16301	vtn	HOST_MOVED	$\begin{aligned} DP_{in} &\xrightarrow{\text{inPacket()}} API_READ \xrightarrow{\text{hostDetected()}} HOST_ADDED \xrightarrow{\text{provider.host.InternalHostProvider}} API_WRITE \xrightarrow{\text{vtn.InternalHostListener}} forward() \xrightarrow{DP_OUT} DP_{out} \end{aligned}$
10	CVE-2019-16302	evpnopenflow	HOST_MOVED	
11	CVE-2019-16302	evpnopenflow	HOST_UPDATED	$\begin{aligned} DP_{in} &\xrightarrow{\text{inPacket()}} API_READ \xrightarrow{\text{provider.host.InternalHostProvider}} API_WRITE \xrightarrow{\text{hostDetected()}} HOST_ADDED \\ &\xrightarrow{\text{evpnopenflow.InternalHostListener}} forward() \xrightarrow{DP_OUT} DP_{out} \end{aligned}$
12	CVE-2019-16297	p4tutorial	HOST_MOVED	
13	CVE-2019-16297	p4tutorial	HOST_REMOVED	$\begin{aligned} DP_{in} &\xrightarrow{\text{inPacket()}} API_READ \xrightarrow{\text{provider.host.InternalHostProvider}} API_WRITE \xrightarrow{\text{hostDetected()}} HOST_ADDED \\ &\xrightarrow{\text{p4tutorial.InternalHostListener}} API_WRITE \xrightarrow{\text{applyFlowRules()}} DP_OUT \xrightarrow{DP_{out}} \end{aligned}$
14	CVE-2019-16297	p4tutorial	HOST_UPDATED	

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

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

5.7.1.2 Method

The attack occurs in two stages.

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

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

5.7.1.3 Results and implications

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

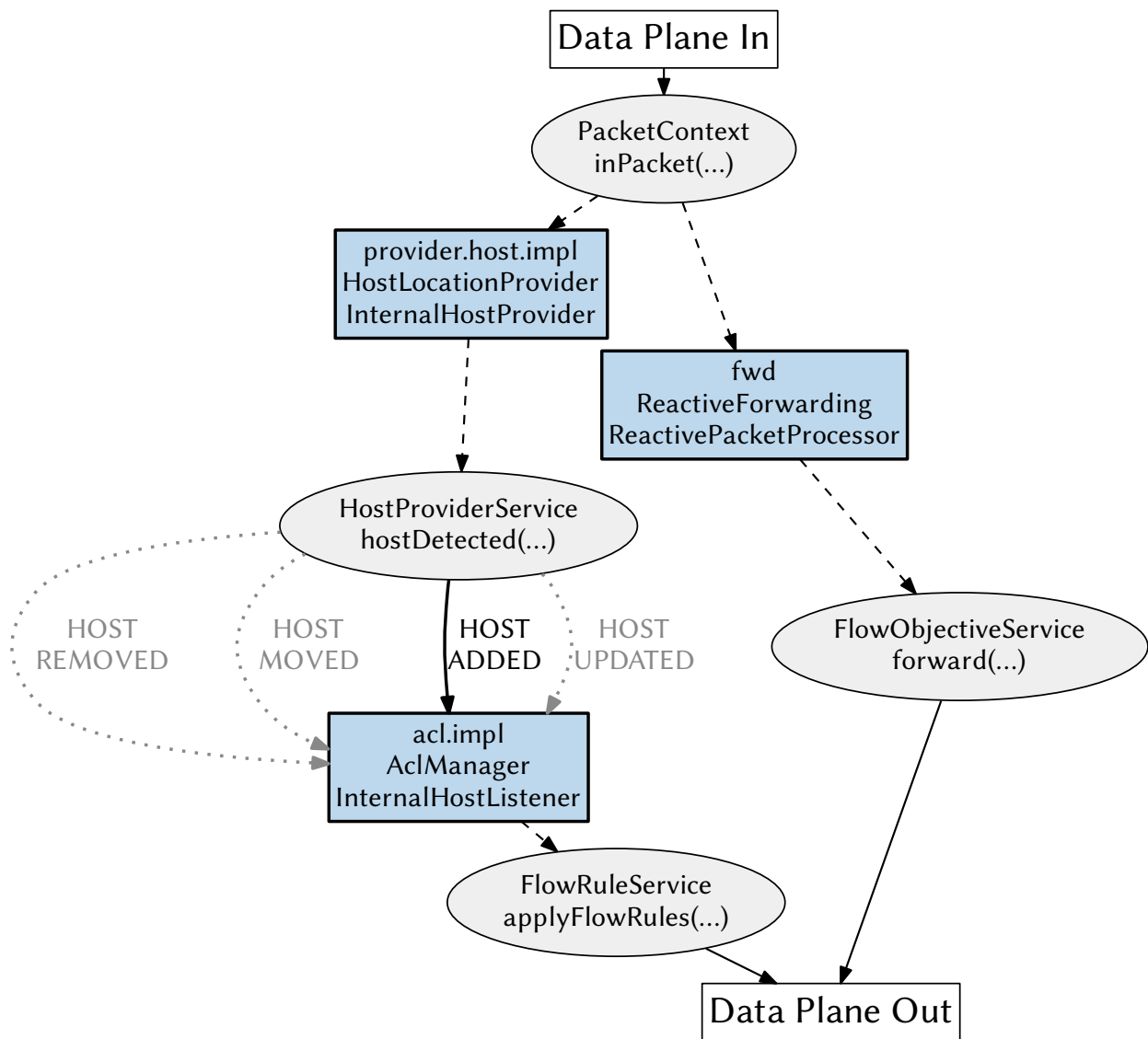


Figure 5.7: Partial event flow graph showing vulnerable code paths used in CVE-2018-12691. Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, bold edges represent event dispatches, and dashed edges represent API calls. (Dotted gray edges represent unhandled event types, which are shown for reference.)

5.7.1.4 Event flow graph

Figure 5.7 shows the partial event flow graph with the relevant code paths used by the attacker. The attack's first stage follows the left-side path, in which the attack corrupts the host information in the HostProviderService. The attack's second stage triggers a `HOST_UPDATED` event type that does not get handled by `acl`'s host event listener; in addition, the attack's second stage succeeds as shown by the right-side path.

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

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

5.7.2.1 Summary

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

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

5.7.2.2 Method

The attack occurs in two stages.

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

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

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

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

5.7.2.3 Results and implications

We wrote an exploit that performed the attack and were able to demonstrate that messages could be sent from the attacker to the target. Although the attack assumed that the attacker controlled two hosts on different connection points, an attacker who initially controls only one host could use the previous exploit in Section 5.7.1 to compromise a second host so as to perform the attack in this section. Much like the exploit in Section 5.7.1, the increased access has significant consequences if our assumptions about the security of data plane access control are incorrect. For instance, if hosts h1 and h2 were segmented and isolated by policy (*e.g.*, to satisfy regulatory compliance requirements), then clever manipulation of host events can effectively bypass such protections.

5.7.2.4 Event flow graph

Figure 5.8 shows the partial event flow graph with the relevant code paths used by the attacker. The attack’s first stage follows the path through the host mobility app, `mobility`, in the figure’s center. The host mobility app responds to the `HOST_MOVED` event type and removes flow rules. The access control app, `acl`, does not handle the `HOST_MOVED` event, and thus the app does not install new flow rules. The attack’s second stage succeeds as shown by the path on the right side of the figure.

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

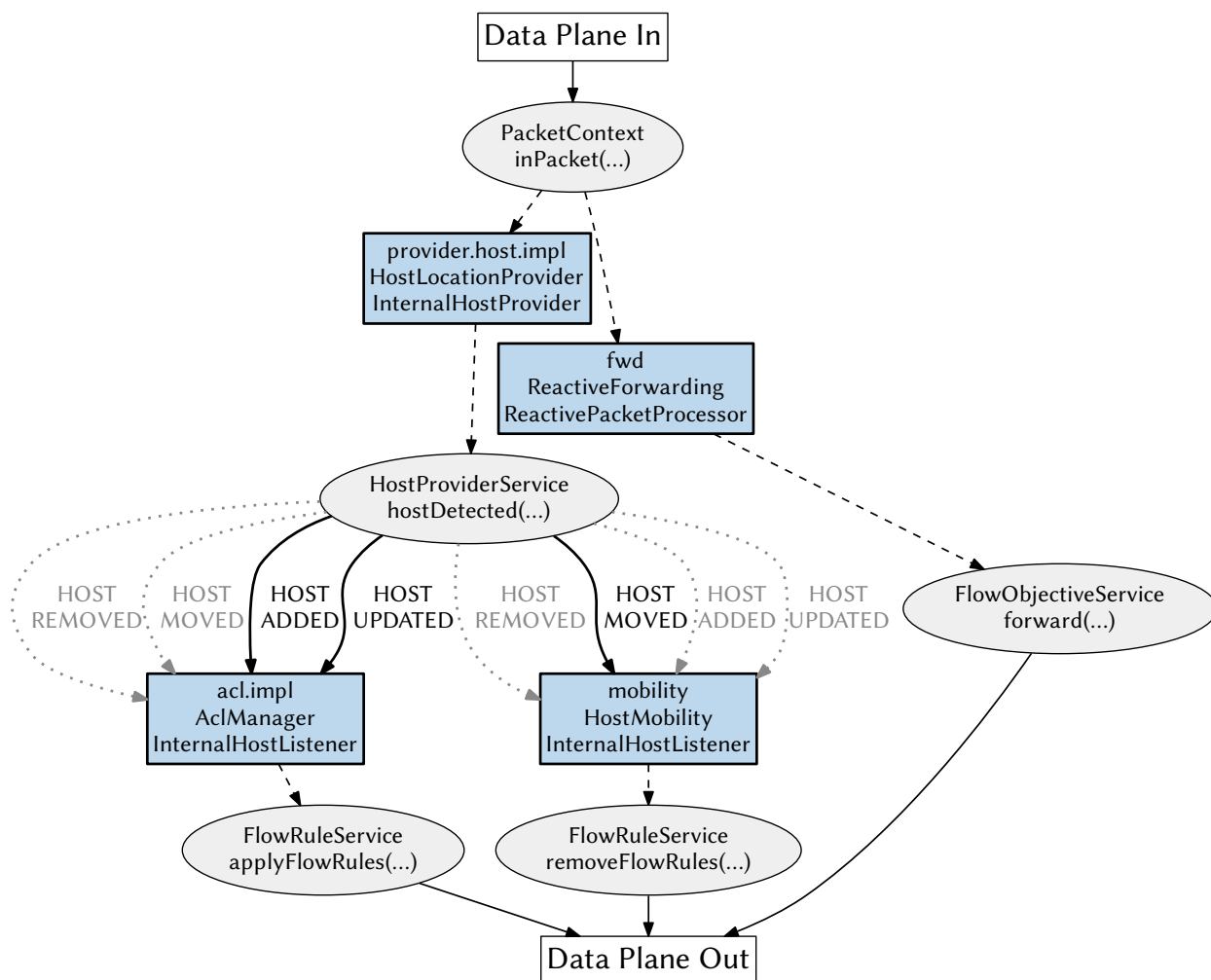


Figure 5.8: Partial event flow graph showing vulnerable code paths used in CVE-2019-11189. Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, bold edges represent event dispatches, and dashed edges represent API calls. (Dotted gray edges represent unhandled event types, which are shown for reference.)

5.7.3 Other Vulnerabilities

In Table 5.1, we summarize the remaining vulnerabilities that EVENTSCOPE discovered, grouped by app.

Vulnerabilities 3–5 (virtualbng) The virtual broadband network gateway app, `virtualbng`, maintains a relationship between a network’s set of private IP addresses and public-facing IP addresses on the Internet [223]. The app also installs network intents, which get translated to new flow rules, to allow the network’s hosts with private IP addresses to connect to the Internet. The app’s host event listener handles the `HOST_ADDED` event type but does not handle the remaining three host event types. As a result, the app does not handle any state updates about the virtual gateways it has previously created if a host changes its information (*e.g.*, new location). A malicious host could spoof that host’s identity, via a process similar to that described in Section 5.7.2.2, to cause `HOST_UPDATED` or `HOST_MOVED` event types to be triggered. Furthermore, when a host is removed, the app does not asynchronously remove its intents (or, by extension, its flow rules) that it previously installed because it does not handle `HOST_REMOVED` event types.

Vulnerabilities 6–8 (mobility) The host mobility app, `mobility`, listens for host-related events and cleans up any related flow rules if a host has moved. Related work [49] has shown how the host mobility app in ONOS can be abused by hosts to force ONOS to reinstall flow rules and cause a control plane denial-of-service attack. Instead, we focus here on the absence of what event types `mobility` handles. The app’s host event listener handles the `HOST_MOVED` event type (as expected) but does not handle the remaining three host event types. If `mobility` is expected by other apps to be responsible for cleaning up flow rules, then a host whose information has been updated (where updating would trigger a `HOST_UPDATED` event type), would not cause a flow removal and might lead to stale flow rules. If there is sufficient time between a moved host’s removal from and addition back into the network, it may trigger a `HOST_REMOVED` event followed by a `HOST_ADDED` event. As `mobility` does not handle either event type, the expected flow removal by `mobility` would not occur.

Vulnerability 9 (vtn) The virtual tenant network app, `vtn`, provisions virtual networks as overlays over physical networks [224]. The app handles all of the host event types except for `HOST_MOVED`. For the host event types that are handled, the app installs flow rules for added hosts (*i.e.*, `HOST_ADDED`), removes flow rules for removed hosts (*i.e.*, `HOST_REMOVED`), and installs and removes flow rules for any host that has changed its properties but not moved (*i.e.*, `HOST_UPDATED`). A host that moves (*i.e.*, `HOST_MOVED`) would not have any actions taken by the app; as a result, flow rules would not be reinstalled, and denial of service could occur.

Vulnerabilities 10–11 (evpnopenflow) The Ethernet VPN app, `evpnopenflow`, uses OpenFlow to install MPLS-labeled overlay routes for virtual private networks [225]. The app’s host event listener handles the `HOST_ADDED` and `HOST_REMOVED` event types, which call functions that are responsible for finding routable paths, installing flow rules, and removing flow rules. The app does not handle hosts moving (*i.e.*, `HOST_MOVED`) or being updated (*i.e.*, `HOST_UPDATED`), and that could cause denial of service to such hosts if old flow rules are not removed and new flow rules are installed.

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

5.8 Discussion

5.8.1 SDN Design Concerns

App composability We found that some apps, which we term “helper apps,” were designed to perform functionality on behalf of other apps currently running. One helper app, `mobility`, removes flow rules when hosts move within the network. However, as we noted with respect to our exploit in Section 5.7.2, if an app’s design does not account for helper apps that are taking actions on its behalf, then the combination of apps may introduce vulnerabilities that arise from a lack of coordinated responsibility. That suggests a need for stronger integration testing among apps; `EVENTSCOPE` is useful in identifying the subsets of apps that may interact.

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

Host migration Although host migration hijacking is a known problem [22, 55, 58, 93], we found that ONOS v1.14.0 and earlier versions do not provide any protections against the broader class of adversarial host-generated data plane input. That suggests a strong cross-plane attack vector, and EVENTSCOPE’s event flow graph can show the extent to which the control plane’s control flow can be altered.

Event abstraction While EVENTSCOPE’s discovered vulnerabilities do relate to host movement, such vulnerabilities differ from the host migration vulnerabilities discovered in related work [22, 55, 58, 93]. Those previously known vulnerabilities specifically use incoming data plane packets to target the host migration service. In contrast, EVENTSCOPE’s discovered vulnerabilities occur one abstraction layer higher: the host migration service declares that a host has moved, and other apps attempt to update their own states to account for such movement. EVENTSCOPE’s discovered vulnerabilities could occur as a result of benign host migration. For example, the acl app relies on a host migration service event (*i.e.*, HostEvent) instead of relying directly on data plane packets because the semantic notion of host migration is a useful abstraction for other apps, too. We believe that future apps will likely follow a similar trend of using abstracted events. One of our goals is to make event propagation more understandable for practitioners and developers. In that context, we believe that EVENTSCOPE’s discovered vulnerabilities are distinct from and complementary to the host migration vulnerabilities found in related work.

Other controllers Much like ONOS’s packet processor, Floodlight’s [28] processing chains allow for specific execution ordering. ONOS contains a more sophisticated, extensive, and distributed event-driven architecture than Floodlight, and we opted to evaluate the more sophisticated architecture. ONOS also contains event processing that does not specify ordering, which is the case for the majority of ONOS event kinds (*i.e.*, all non-packet events). Although the event flow graph captures the ordering of different events (*e.g.*, a packet event that subsequently triggers a host event), the graph does not capture the processing order within an event (*e.g.*, the packet event goes to app X, then app Y).

5.8.2 Limitations

EVENTSCOPE cannot establish the absence of vulnerabilities. NICE [105] shows that a large state space search is needed to reason about the absence of vulnerabilities, but such state does not scale beyond simple apps and controllers. EVENTSCOPE lets developers and practitioners understand complex app interactions using a scalable approach.

To help practitioners identify unsafe operating conditions, EVENTSCOPE can generate contexts

under which certain combinations of apps may manifest a vulnerability; however, EVENTSCOPE does not generate exploits. Automated exploit generation [216] is an ongoing research area, and we consider automated SDN exploit generation to be future work.

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

5.9 Related Work

SDN security Cross-plane attacks have been studied in specific contexts. Yoon *et al.* [80] refer to these attacks as control plane remote attacks for network-view manipulation. SPHINX [93], TOPOGUARD [22], TOPOGUARD+ [55], and SECUREBINDER [58] reveal the lack of protection against link fabrication attacks and host location hijacking. However, none of the four systems analyze the extent to which the untrusted data plane inputs propagate via events to other components in the controller, and such analysis is necessary for cases where apps’ competing behaviors create vulnerabilities.

CONGUARD [109] identifies time-of-check-to-time-of-use race conditions in SDN controllers and provides a generalized model of control plane happens-before relations, but the generalized semantics do not account for more sophisticated app semantics whose incomplete event handling can be exploited.

INDAGO [114] and SHIELD [113] use static analysis to analyze SDN apps and summarize their API use. INDAGO proposes machine learning techniques to determine whether an app is malicious or benign based on its sources and sinks from API call use. Given that benign apps can be co-opted by other apps as confused deputies [51], we find the distinction of malicious and benign labeling to be irrelevant for EVENTSCOPE. Instead, EVENTSCOPE approaches the problem from a global event dependency view.

Event-driven architectures We consider the SDN architecture *vis-à-vis* Android and Web browser extensions. SDN and Android differ based on the mechanisms by which data are passed and on how apps coordinate with each other [51]. Event-driven SDN relies on a central event dispatching mechanism over a limited set of network events, which implies that SDN apps must coordinate with each other to apply policies to and to enforce security over the shared data plane resource. Vulnerability tools and analyses for Android [196, 214, 226, 227, 228, 229, 230, 231] and browser ex-

tensions [232, 233, 234, 235] have focused primarily on preventing information leakage among apps or extensions rather than specifically on how unhandled events affect global control flow.

Vulnerability discovery Livshits and Lam [215] secure Java programs from unchecked Web-based input vulnerabilities. We study the analogous SDN problem of untrusted data plane input and model our attacks using a two-stage model of initial injection and subsequent manipulation. Yang *et al.* [236] note the challenges for event-driven callbacks in Android, which we consider in our SDN component model. Monperrus and Mezini [237] study the use of missing method calls as indicators of deviant code, using an approach similar to that used for the unhandled event type problem. CHEX [214] identifies entry points in Android applications and uses “app splitting” to identify all code that is reachable from a given entry point. We adapted CHEX’s notion of app splitting for use in building event flow graphs. Code property graphs [211] combine abstract syntax trees, control flow graphs, and program dependence graphs into a unified data structure for automated vulnerability discovery, but have scalability concerns.

Network debugging Cross-plane and cross-app attacks can be tracked using causality tracking and data provenance approaches. PROVSDN [51] prevents cross-app poisoning attacks in real time using a provenance graph structure to enforce information flow control, and FORENGUARD [129] records previous causal relationships to identify root causes. Negative provenance [131], differential provenance [202], and meta provenance [132] have been proposed to explain why SDN routing events did not occur and to propose bug fixes, but such methods require either a history of traces or reference examples of “good” behavior; furthermore, analysis of SDN applications in those systems must be written in or translated into the Datalog language NDlog prior to analysis. The aforementioned systems record code paths that were taken rather than all potential code paths, which limits their effectiveness in identifying potential vulnerabilities ahead of time.

DELTA [112], BEADS [111], and STS [110] use fuzzing to generate data plane inputs, but the space of potential inputs is complex for large and complex event-driven controllers. NICE [105] models basic control plane semantics (*e.g.*, flow rule installation ordering) and uses the generated state space to perform concrete symbolic (*i.e.*, concolic) execution to find bugs; however, even for simple single apps, the approach does not scale well. VeriFlow [96] uses network correctness properties to prevent flow rules from being installed. However, such approaches require a formal statement about app behavior. Given that the checks occur in the southbound API, such tools do not identify the sources of vulnerabilities.

5.10 Conclusion

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

CHAPTER 6

CONTROL PLANE CAUSAL ANALYSIS

Software-defined networking (SDN) has emerged as a flexible network architecture for central and programmatic control. Although SDN can improve network security oversight and policy enforcement, ensuring the security of SDN from sophisticated attacks is an ongoing challenge for practitioners. Existing network forensics tools attempt to identify and track such attacks, but holistic causal reasoning across control and data planes remains challenging.

We present PicoSDN, a provenance-informed causal observer for SDN attack analysis. PicoSDN leverages fine-grained data and execution partitioning techniques, as well as a unified control and data plane model, to allow practitioners to efficiently determine root causes and to make informed decisions on mitigating attacks. We implement PicoSDN on the popular ONOS SDN controller. Our evaluation across several attack case studies shows that PicoSDN is practical for the identification, analysis, and mitigation of SDN attacks.

6.1 Introduction

Over the past decade, the software-defined networking (SDN) architecture has proliferated as a result of its flexibility and programmability. The SDN architecture decouples the decision-making of the *control plane* from the traffic being forwarded in the *data plane*, while logically centralizing the decision-making into a *controller* whose functionality can be extended through *network applications* (or *apps*).

SDN has been touted as an enhancement to network security services, given that its centralized design allows for complete oversight into network activities. However, the programmable nature of SDN creates new security challenges and threat vectors. In particular, the control plane’s state and functionality can be maliciously influenced by data input originating from the data plane and apps. These *cross-plane* [22, 43, 44, 48, 55, 93] and *cross-app* [45, 51] attacks have significant security repercussions for the network’s behavior. An adversary only needs to attack data plane hosts or apps, and does not have to compromise the controller. Complex SDN control planes and new attack vectors make a network security practitioner’s job more challenging, as he or she must quickly collect any evidence of an attack, establish possible root causes, and mitigate such causes to prevent future

attacks.

Network causality and provenance tools are becoming more popular as evidenced by recent research contributions [51, 129, 131, 141]. However, we argue that such tools have limitations in terms of providing precise and holistic causal reasoning.

First, the control plane’s causality (or provenance) model has a significant effect on the precision with which a practitioner can identify root causes. If the control plane’s data structures are too coarse-grained or if the control plane uses long-running processes, this can lead to *dependency explosion* problems in which too many objects share the same provenance. That reduces the ability to identify precise causes.

Second, the control plane’s decisions cause the data plane’s configuration to change; the effects of the data plane’s configuration on packets sent to the controller cause subsequent control plane actions. When network causality and provenance tools examine the control plane alone, the indirect causes of control plane actions that result from data plane packets will lead to an *incomplete dependency* problem that ignores the data plane topology.

Third, a practitioner will want to know not only the root causes for an action but also the extent to which such root causes impacted other network activities. For instance, if a spoofed packet is found to be the attack vector for an attack, then the practitioner will want to investigate what else that spoofed packet influenced to understand whether other attacks and undesirable behavior have also occurred.

Overview We present PICO SDN, a tool for SDN attack analysis that mitigates the aforementioned dependency explosion and incomplete dependency challenges. PICO SDN allows practitioners to effectively and precisely identify root causes of attacks. Given evidence from an attack (*e.g.*, violations of intended network policies), PICO SDN determines common root causes in order to identify the extent to which those causes have affected other network activities.

PICO SDN’s approach uses *data provenance*, a data plane model, and a set of techniques to track the lineage of objects, activities, and system principals. PICO SDN records provenance graphically to allow for efficient queries over past state. Although similar network forensics tools have also used graphical structures [51, 129, 130], these tools’ provenance models suffer from dependency explosion or incomplete dependency problems. To account for those challenges, PICO SDN performs fine-grained partitioning of control plane data objects and leverages the loop-based event listeners common in SDN apps to further partition data and process execution, respectively. PICO SDN also incorporates the data plane’s topology such that indirect control plane activities caused by data plane packets are correctly encoded, which mitigates incomplete dependencies. Finally, PICO SDN’s toolkit reports the impacts of suspected root causes, identifies how network identifiers (*i.e.*, host identities) evolve over time, and summarizes the network’s flow rule configuration.

We have implemented PICO SDN within the popular ONOS SDN controller [25]. Many telecommunications providers, such as Comcast, use ONOS or one of its proprietary derivatives [159]. We evaluated PICO SDN by executing and analyzing recent SDN attack scenarios found in the literature and in the Common Vulnerabilities and Exposures (CVE) database. PICO SDN precisely identifies the root causes of such attacks, and we show how PICO SDN’s provenance model provides better understanding than existing network tools do. Our implementation imposes an average overhead latency increase of between 7 and 21 ms for reactively instantiated flow rules (with the increase determined by topology size), demonstrating PICO SDN’s practicality in realistic network settings.

Contributions Our main contributions are:

1. An approach to the **dependency explosion problem for SDN attack provenance** that utilizes event listeners as units of execution.
2. An approach to the **incomplete dependency problem for SDN attack provenance** that incorporates a data plane model and tracking of network identifiers.
3. The **design and implementation** of PICO SDN, which we use with ONOS to evaluate SDN attacks and to demonstrate PICO SDN’s causal analysis benefits.
4. The **performance and security evaluations** of PICO SDN on recent SDN attacks.

6.2 Challenges

Many real-world SDN attacks leverage data plane dependencies and long-running state corruption tactics to achieve their goals. SDN controllers are susceptible to attacks from data plane hosts that poison the controller’s network state view and cause incorrect decisions [22, 44, 48, 55, 93, 238]. We consider a motivating attack to illustrate the limitations that a practitioner encounters when using existing network forensics tools.

6.2.1 Motivating Attack Example

Scenario Consider the control plane attack CVE-2018-12691 [44, 238] in ONOS. It enables an attacker to use spoofed packets to circumvent firewall rules. This class of cross-plane attack leverages spoofed data plane input to fool the controller into maliciously changing the data plane forwarding. Complete prevention of such attacks is generally challenging, as spoofed information from data plane hosts is a notorious network security problem in SDN [22, 58, 93]. Such attacks can also be

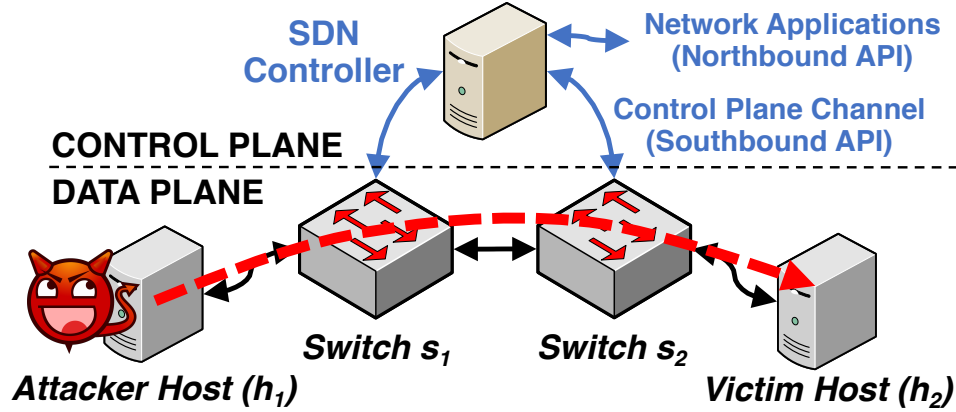


Figure 6.1: Topology of the CVE-2018-12691 attack scenario described in Section 6.2.1. The red path represents the attacker’s desired data plane communication from h_1 to h_2 .

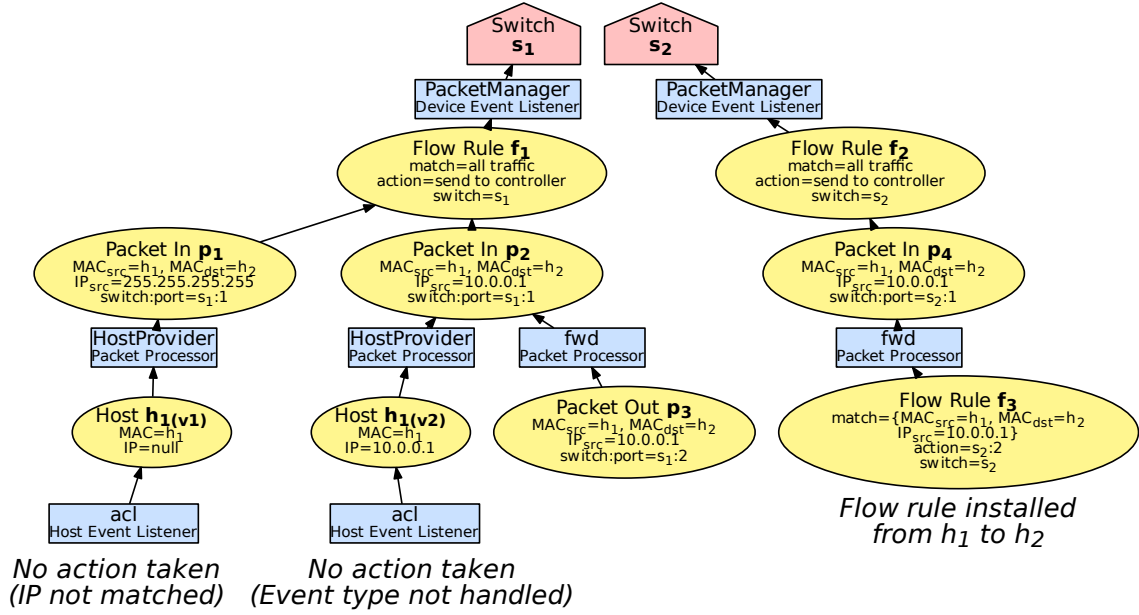
one part of a multi-stage attack in which the attacker’s goal is to defeat the data plane access control policy and move laterally across data plane hosts to gain additional access [4].

Suppose that the attack is carried out on a network topology as shown in Figure 6.1. Assume that the controller runs a data plane access control application and a reactive¹ forwarding application. The attack works as follows. A malicious data plane host, h_1 , wants to connect to a victim host, h_2 , but the data plane access control policy is configured to deny traffic from h_1 to h_2 based on its IP address. The malicious host h_1 emits into the data plane a spoofed ICMP packet, p_1 , with an invalid IP address. The controller creates a host representation object for h_1 with a valid MAC address but no IP address. The data plane access control application, *acl*, checks to see if it needs to insert new flow rules based on the data plane access control policy. As the controller does not associate h_1 with an IP address, no flow rules are installed.

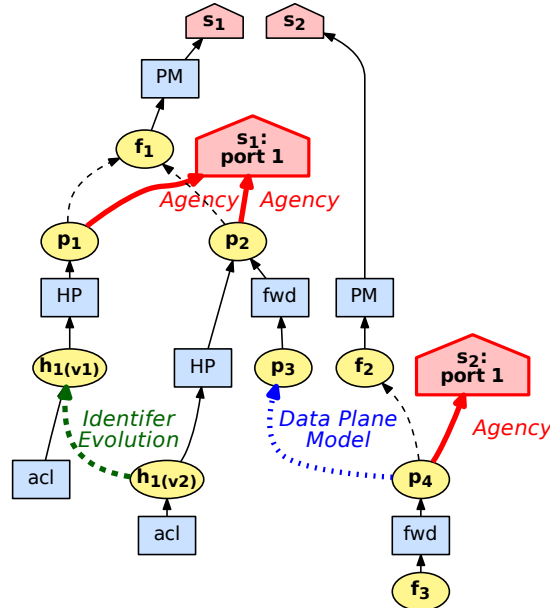
Some time later, h_1 sends to h_2 a packet, p_2 , with a valid source IP address. ONOS updates the host object for h_1 with h_1 ’s actual IP address. Unfortunately, at this point, a bug causes the data plane access control application to not handle events in which a host object is updated. Thus, the update never triggers the application to install flow deny rules that prevent h_1 from sending traffic to h_2 . The result is that the reactive forwarding application forwards the packet out (*i.e.*, p_3).

Investigation At a later point in time, a practitioner discovers that data from h_2 have been exfiltrated out of the network through connections originating in h_1 that violate the intended data plane access control policy. To investigate why the policy was bypassed, the practitioner attempts to perform causal analysis using a *provenance graph* (depicted in Figure 6.2a) over the control plane’s past state.

¹Although we discuss a reactive SDN configuration here as an example, PicoSDN’s design generalizes to proactive SDN configurations, too. We refer the reader to Section 6.8 for further discussion.



(a) Relevant provenance generated based on techniques from prior work [129]. The activities from switches s_1 and s_2 appear to be independent of each other, masking the derivation of a root cause of s_2 's flow rule f_3 from host h_1 's activities on switch s_1 .



(b) Relevant provenance generated with PICO SDN, which includes a data plane model, network identifiers, and precise agency.

Figure 6.2: Data, process, and agency provenance of the CVE-2018-12691 attack scenario described in Section 6.2.1. Ellipses represent SDN control plane data objects, rectangles represent SDN processes, and pentagons represent SDN principal agents.

The practitioner knows that a flow rule or a set of packets between h_1 and h_2 must have allowed the communication, so he or she starts with a piece of *evidence* that consists of the flow rule installed by the reactive forwarding application that allowed traffic from h_1 to h_2 on switch s_2 (i.e., flow rule f_3). The practitioner issues a query and identifies a set of possible root causes related to the lineage of that flow rule.

6.2.2 Existing Tool Limitations

However, the practitioner runs into several challenges when using existing tools to generate a graph such as the one in Figure 6.2a. Although linking h_1 's packets to s_1 's default flow rule (i.e., f_1) does capture past causality, the practitioner is easily overwhelmed when *all* packets over all time from any of s_1 's ports are also linked to that default flow rule. The practitioner also finds that switches s_1 and s_2 as principal agents become too coarse-grained to enable pinpointing of attribution. Since existing tools do not account for the data plane as a causal influence, the result in Figure 6.2a is a set of two disconnected subgraphs. That disconnection prevents the practitioner from performing a meaningful backward trace. Finally, backward tracing alone would not provide the practitioner with details about the attack's other effects. We generalize those challenges and consider them in depth below.

Limitation (L1): Dependency explosion Identification of provenance suffers from the *dependency explosion problem* in which long-running processes or widely used data structures within a system can create false dependencies. That problem can be mitigated through *data partitioning* or *execution partitioning*. SDN tools, such as CAP [51] and FORENGUARD [129], mitigate the problem, but limitations remain. For instance, PROVSDN's API-centric model would create many false dependencies among an app's event listener instances because an API call would be falsely dependent on all previous API calls. FORENGUARD's event-centric model uses execution partitioning, but if we apply it as shown in Figure 6.2a, we see that a controller that installs default flow rules (i.e., f_1) will cause all unmatched packets (i.e., p_1 and p_2) to become dependent on it. As a result, FORENGUARD's modeling approach can suffer from data partitioning challenges when too many unrelated effects of a root cause must also be analyzed.

Limitation (L2): Coarse-grained agency and false attribution A similar challenge exists in the assignment of data plane agency. In Figure 6.2a, the agency traces back to a switch, either s_1 or s_2 . Although this correctly implies that one of the root causes of the attack is s_1 or s_2 , it is not a particularly useful insight because all other activities have one of these root causes, too. Instead, should the responsibility be assigned to a notion of a host? Given that network identifiers (e.g., MAC addresses)

are easily spoofable, assigning agency to hosts would not solve the problem either; malicious hosts would simply induce false dependencies in the provenance graph.

Limitation (L3): Incomplete dependencies In contrast to false dependencies, the *incomplete dependencies* occur when the provenance model does not capture enough information to link causally related activities. For SDN attacks, that occurs when the data plane’s effects on the control plane are not captured by an implicit *data plane model*. In our attack scenario in Section 6.2.1, the reactive forwarding application reacts to activities from switch s_1 before forwarding the packet (*i.e.*, p_3) out to other ports. On the other end of one of s_1 ’s ports, switch s_2 receives that incoming packet (*i.e.*, p_4) and further processes it. Figure 6.2a’s disconnected subgraphs appear to show that switch s_1 ’s history of events is independent of switch s_2 ’s history of events. Thus, if a practitioner were starting his or her investigation from a flow rule on switch s_2 , he or she would not be able to see that the root cause occurs because of earlier events related to switch s_1 and the malicious host h_1 ’s spoofed packets. PROVSDN and FORENGUARD do not account for this data plane model and would thus suffer from incomplete dependencies. Other tools [131, 132, 202, 239] model the implicit data plane, but are applicable only in the declarative networking paradigm. Most of the popular SDN controllers [25, 28, 30], in contrast, use an operating-system-like imperative paradigm.

Limitation (L4): Interpretation and analysis Even if the dependency-related challenges previously described were mitigated, it can still be challenging to interpret provenance graphs. For instance, if the practitioner in our attack scenario from Section 6.2.1 wanted to understand how network identifier bindings (*e.g.*, the network’s bindings between a host’s MAC address and its location in the data plane) changed over time, the provenance graph in Figure 6.2a would not support that; it does not directly link the host objects because their generation were not causally related.

PROVSDN and FORENGUARD use *backward tracing* to start with a piece of evidence and find its information flow ancestors or set of root causes, respectively. However, if the practitioner wanted to know the other effects of the spoofed packet generated by h_1 , that analysis would require *forward tracing* techniques that start at a cause and find its progeny to determine what other data and processes were affected. As neither PROVSDN nor FORENGUARD performs forward tracing, the practitioner would not be able to discover other relevant unexpected artifacts of the attack, such as acl’s failure to generate flow deny rules.

The practitioner ultimately wants to answer network connectivity questions of the form “Which packet(s) caused which flow rule(s) to be (or not to be) installed?” However, the SDN controller’s event-based architecture can be itself complex [44]. Although the complexity must be recorded to maintain the necessary dependencies, most of the complexity can be abstracted away to answer a practitioner’s query. Thus, abstracted *summarization* is necessary for practitioners to understand

attacks easily and quickly.

6.2.3 Our Approach

Motivated by the attack presented in Section 6.2.1 and the previous tools' limitations noted in Section 6.2.2, we highlight how PICO SDN would mitigate the issues. PICO SDN uses a provenance model that accounts for data and execution partitioning with precise agency, while also incorporating the implicit data plane effects on the control plane (Section 6.3). PICO SDN also provides techniques to aid in analysis (Section 6.5).

Applying PICO SDN produces the graph shown in Figure 6.2b. Rather than relying solely on the default flow rule f_1 as a root cause, the practitioner can see that packets p_1 and p_2 originate at a host on switch s_1 's port 1 (**L1**). That also allows the practitioner to precisely identify agency at the switch port (rather than switch) level (**L2**). The previously independent activities from each switch are linked by the data plane model that connects p_4 with p_3 (**L3**), which allows the practitioner to backtrack from s_2 to s_1 (**L4**). Finally, the practitioner can see how host h_1 's network identifier information evolved over time (**L4**) and can summarize the past network state (**L4**).

6.3 PICO SDN Provenance Model

In order to reason about past activities and perform causal analysis, we first define a *provenance model* that formally specifies the relevant data, processes and principal identities involved in such data's generation and use.² Our unified approach accounts for app, control, and data plane activities, which allows us to reason holistically about SDN attacks.

6.3.1 Definitions

Provenance graph A *provenance graph*, denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a directed acyclic graph (DAG) that represents the lineages of objects comprising the shared SDN control plane state. Informally stated, the graph shows all of the relevant processes and principal identities that were involved in the use or generation of such control plane objects. We use the graph to analyze past activities to determine root causes (*i.e.*, backward tracing) and use those root causes to determine other relevant control plane activities (*i.e.*, forward tracing).

²Our model is loosely based on the W3C PROV data model [51, 184, 186].

Table 6.1: Nodes in the PICO SDN Provenance Graph Model.

Node class	Node meaning and node subclasses
Entity	A data object within the SDN control plane state, used or generated through API service calls or event listeners <i>Subclasses:</i> Host, Packet (<i>subsubclasses:</i> PacketIn, PacketOut), FlowRule, Objective, Intent, Device, Port, Table, Meter, Group, Topology, Statistic
Activity	An event listener or a packet processor used by an SDN app or controller <i>Subclasses:</i> EventListener, PacketProcessor
Agent	An SDN app, an SDN controller core service, a switch port, or a switch (<i>i.e.</i> , device) <i>Subclasses:</i> App, CoreService, SwitchPort, Switch

Table 6.2: Edges (Relations) in the PICO SDN Provenance Graph Model.

Valid edge (relation) class	Relation meaning
<i>Entity wasGeneratedBy Activity</i>	Creation of an SDN control plane state object
<i>Activity used Entity</i>	Use of an SDN control plane state object
<i>EventListener used Entity</i>	An event listener’s use of the SDN control plane state object
<i>PacketProcessor used Packet</i>	A packet processor’s use of a data plane packet
<i>Entity wasInvalidatedBy Activity</i>	Deletion of a data object within the SDN control plane state
<i>Entity wasDerivedFrom Entity</i>	Causal derivation of one SDN control plane state object to another object
<i>PacketIn wasDerivedFrom FlowRule</i>	Causal derivation of an incoming packet based on a previously installed flow rule (<i>e.g.</i> , default flow rule)
<i>PacketIn wasDerivedFrom PacketOut</i>	Causal derivation of an incoming packet from one switch based on the outgoing packet of another switch
<i>Entity wasRevisionOf Entity</i>	Non-causal revision (<i>i.e.</i> , new version) of an SDN control plane state object
<i>Activity wasAssociatedWith Agent</i>	Agency or attribution of an SDN control plane event
<i>Packet wasAttributedTo SwitchPort</i>	Agency or attribution of a data plane packet with the respective switch port on which the packet was received

Nodes Each node $v \in \mathcal{V}$ belongs to one of three high-level classes: Entity, Activity, and Agent. Each high-level node class is explained with its respective subclasses in Table 6.1. We detail the design choices and semantics of these nodes in Section 6.3.2. A node may also contain a dictionary of key–value pairs.

Edges (relations) Each edge (or relation) $e \in \mathcal{E}$ belongs to one of the classes listed in Table 6.2; rows that are indented show relations that have more precise subclasses and meanings than their

superclass. Relations form the connections among the control plane objects, the network activities involved in their generation and use, and principal identities within the SDN components.

Paths (traces) A *backward trace path*, denoted by $t_b = \langle v_0 \rightarrow e_0 \rightarrow \dots \rightarrow e_i \rightarrow v_j \rangle, e_0 \dots e_i \in \mathcal{E}_{\text{class}=\text{wasRevisionOf}}, v_0 \dots v_j \in \mathcal{V}$, is a path of alternating nodes and edges that begins at a node of interest v_0 and ends at an ancestry node v_j . An ancestry node is a predecessor of a node of interest. Given that \mathcal{G} is a DAG, nodes v_1, \dots, v_{j-1} are also ancestry nodes. A backward trace does not include any `wasRevisionOf` edges because such edges represent non-causal relations. A *revision trace path*, denoted by $t_r = \langle v_0 \rightarrow e_0 \rightarrow \dots \rightarrow e_i \rightarrow v_j \rangle, e_0 \dots e_i \in \mathcal{E}_{\text{class}=\text{wasRevisionOf}}, v_0 \dots v_j \in \mathcal{V}$, is a path of edges that begin at a node of interest v_0 and show the revisions of that node's object starting from an earlier revision node v_j . These revisions are non-causal and are used to identify changes to objects over time.

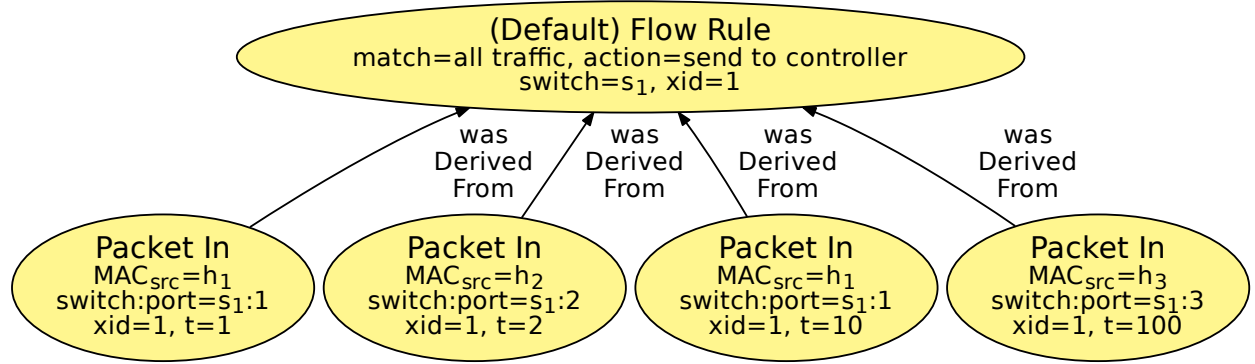
6.3.2 Model Design Choices

Given the aforementioned definitions, we now discuss the design decisions we made in PICO SDN's provenance model. We show how these decisions were influenced by the limitations found in previous work and how these decisions help us solve the challenges outlined in Section 6.2.2.

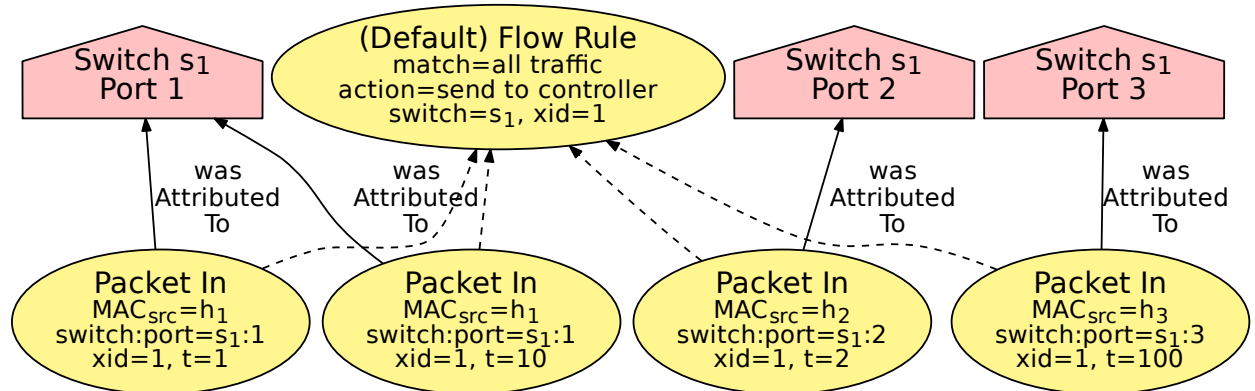
Data and execution partitioning We achieve data partitioning with Entity objects by partitioning the data objects specified in the controller's API. For instance, the ONOS controller's host core service provides the API call `getHosts()`, which returns a set of Host objects. Thus, a natural way to partition data is to identify each Host object as a data partition. The Entity subclasses are generalizable to common SDN control plane state objects as found in the representative ONOS [25], OpenDaylight [30], and Floodlight [28] SDN controllers.

Default flow rules can generate dependency explosions because any incoming packet that does not match other flow rules is sent to the controller for processing. All previously unseen packets become causally dependent on a generalized default flow rule, as shown in Figure 6.3a. To mitigate that problem, our model links any such packets to the respective edge ports that generated the packets, as shown in Figure 6.3b.

We achieve execution partitioning with Activity objects by partitioning each execution of recurring event listeners and packet processors into separate activities. Figure 6.4 shows the differences between API-based modeling and event-based modeling. With event-based modeling, we can more clearly show which Entity objects were used, generated, or invalidated by a given Activity and mitigate the dependency explosion.



(a) Data dependency explosion using default flow rules (used in PROVSDN and FORENGUARD). All packets from switch s_1 that do not match any other flow rules become causally dependent on the default flow rule, which leads to dependency explosion.



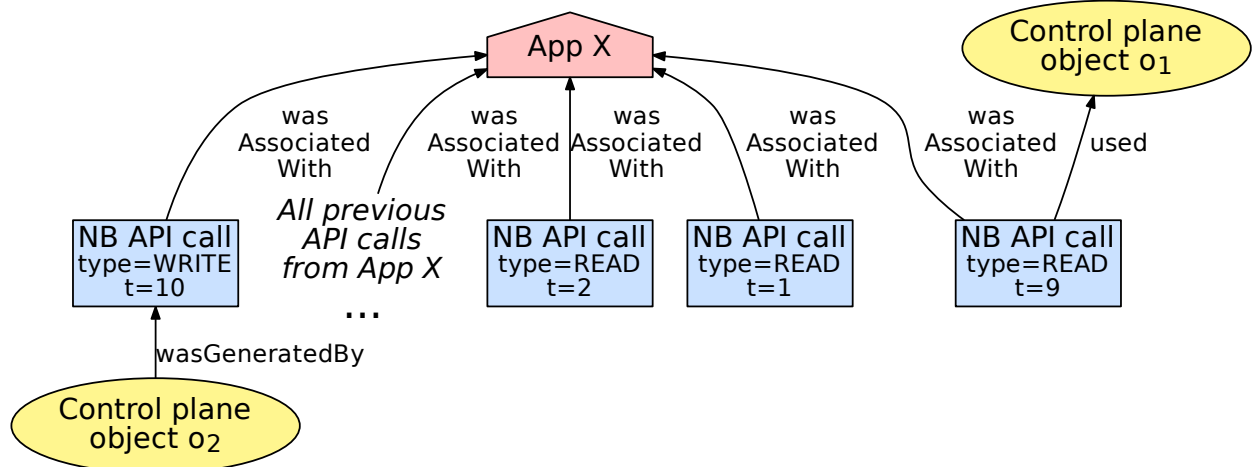
(b) Data partitioning using packets and switch port agents (used in PICO SDN). All packets per switch port are logically grouped together.

Figure 6.3: Data partitioning models for flow rules. Ellipses represent Entity nodes, and house polygons represent Agent nodes.

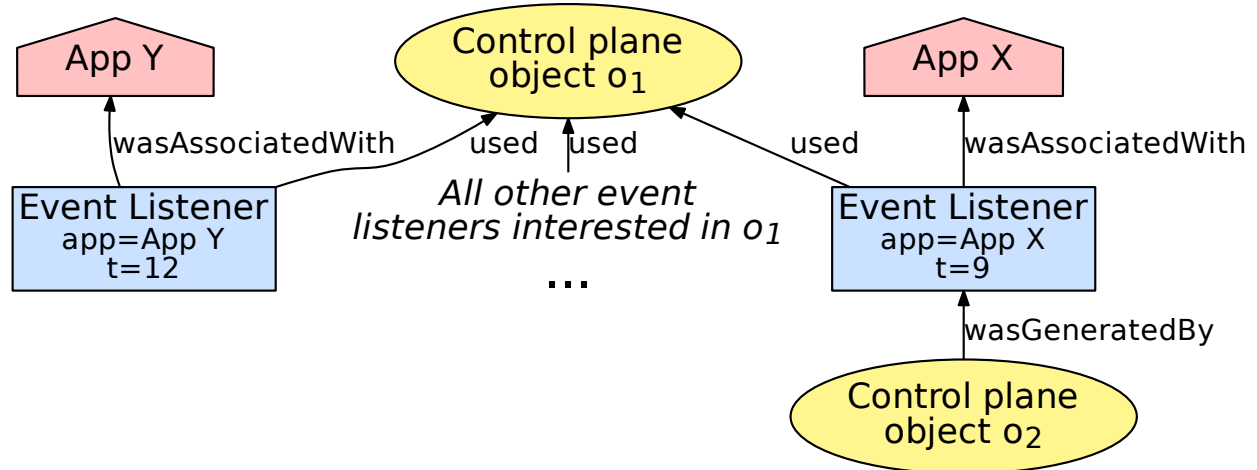
Event listening SDN controllers dispatch events to event listeners. In ONOS, for example, the host service dispatches a HostEvent event (with the corresponding Host object) to any HostEvent listener. We model an event's data object as an Entity node that was used by EventListener nodes, with each event listener invocation represented as its own node.

Data plane model Figure 6.5 shows a diagram of data plane activities between two switches, s_1 and s_2 . Figure 6.5a shows the temporal order of a control plane activity (*i.e.*, generation of an outgoing data plane packet), followed by a data plane activity (*i.e.*, transmission of a data plane packet), followed by another control plane activity (*i.e.*, processing of an incoming data plane packet). As shown in Figure 6.5b, a provenance model without the implicit causality of the data plane shows two separate subgraphs, which makes it impossible to perform a causally meaningful backward trace.

To mitigate that problem, we use a *data plane model* that includes the network's topology and re-



(a) API-based modeling (used in ProvSDN). If one is tracing o_2 's provenance via the API write at time $t = 10$, it will not be clear that *only* the API read of o_1 at $t = 9$ is causally associated with o_2 . The other API reads at $t = 1$ and $t = 2$ represent false dependencies.

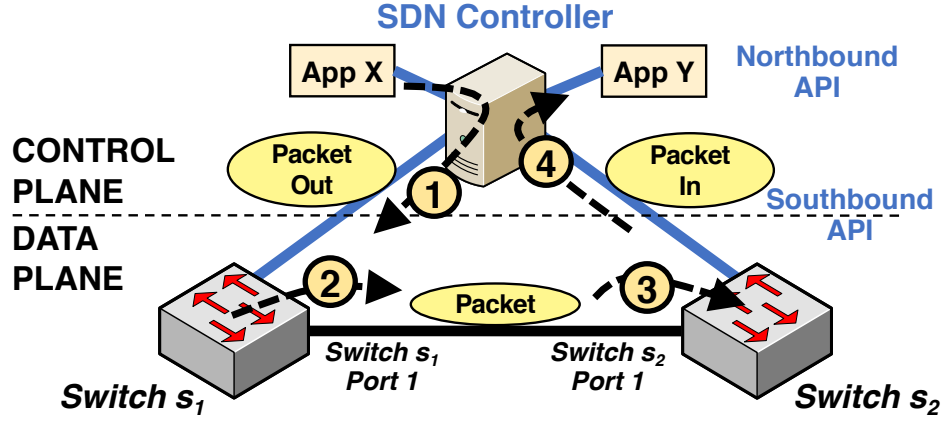


(b) Event-based modeling (used in PicoSDN). If one is tracing o_2 's provenance via the event listener, it will not be clear that o_2 is causally associated with o_1 through App X's event listener.

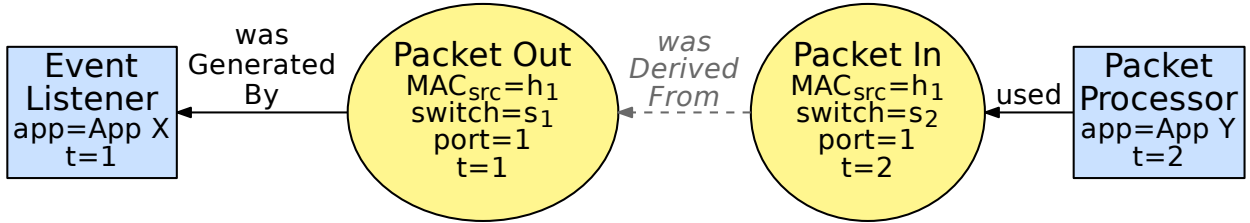
Figure 6.4: Comparison of execution partitioning models. Ellipses represent Entity nodes, rectangles represent Activity nodes, and house polygons represent Agent nodes.

lated happens-before relationships among activities. Our provenance model includes a data-plane-based causal derivation in the relation *PacketIn* *wasDerivedFrom* *PacketOut* to represent the causality.

Network identifiers Control plane objects generated from data plane hosts pose a unique attribution challenge. Data plane hosts can spoof their principal identities, or *network identifiers*, relatively easily in SDN [58] as a result of network protocols (e.g., the Address Resolution Protocol) that do not provide authentication and SDN controller programs that naïvely trust such information [44].



(a) Control plane → data plane → control plane activity. **1:** App X instructs the controller to emit a data plane packet from switch s_1 . **2:** Switch s_1 emits the data plane packet on its link towards switch s_2 . **3:** Switch s_2 receives the incoming data plane packet and sends it to the controller. **4:** App Y processes the data plane packet.



(b) Resulting control plane provenance graph. The dashed edge represents the provenance if we include a data plane model. Without the edge (and the data plane model), the PacketIn from s_2 would not appear to be causally dependent on PacketOut from s_1 ; that represents an incomplete dependency.

Figure 6.5: Data plane model.

Ideally, each data plane host would have its own principal identity, but that is impossible if hosts can spoof their network identifiers.

To mitigate that problem, our provenance model offers two features: *edge ports as principal identities* and *network identifier revisions*. To enable those abilities, we model each edge port³ as a principal identity, or Agent node; Figure 6.3b shows an example. As we assume in our threat model (described in detail in Section 6.4) that switches are trusted, we can trust that the data plane traffic originating at a particular switch port is actually originating at such port. Whether or not a host claiming to have a particular identifier (e.g., MAC address) on that port is legitimately located on that port cannot be verified from the data plane alone. To account for that, we model identifier changes by using the non-causal relation *wasRevisionOf*. It allows for a succinct trace of identifier changes over time.

³As opposed to an internal port that links a switch with another switch.

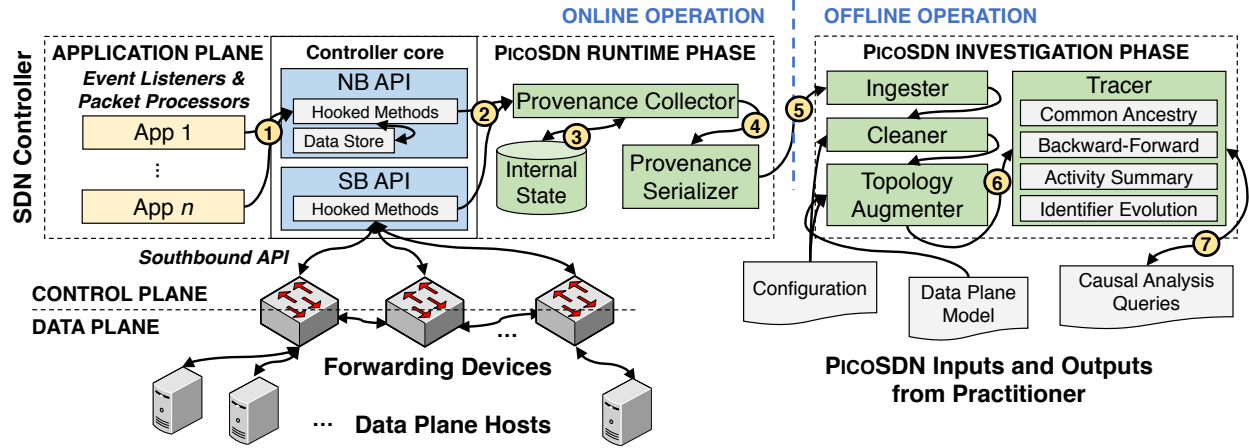


Figure 6.6: PICO SDN architecture overview with example workflow. **1:** An app makes an API call. **2:** PICO SDN’s API hooks register the API call. **3:** The provenance collector checks its internal state and makes changes based on the API call. **4:** The provenance serializer generates the relevant graph. **5:** The ingester, cleaner, and topology augmenter prepare the graph. **6:** The tracer receives the graph. **7:** The tracer answers causal analysis queries based on the graph.

6.4 PICO SDN Threat Model

We assume that the SDN controller is trusted but that its services and functionality may be subverted by apps or by data plane input, which is similar to a threat model found in related work [51, 129]. Attackers will try to influence the control plane via *cross-app* poisoning attacks [51] or via *cross-plane* poisoning attacks [22, 48, 55, 93]. As a result, we assume that all relevant attacks will make use of the SDN controller’s API service calls, event dispatches, or both.

We further assume that switches and apps maintain their own principal identities and cannot spoof their identifiers, and indeed we can enforce that policy using public-key infrastructure (PKI) [15]. However, we assume that data plane hosts *can* spoof their network identifiers (e.g., MAC address).

6.5 PICO SDN Design

Based on the provenance model described in Section 6.3, we now present the design of **provenance-informed causal observation** for **software-defined networking**, or PICO SDN. PICO SDN provides fine-grained data and execution partitioning to aid in the identification of SDN attack causes. PICO SDN’s analysis capabilities allow a practitioner to identify evidence of malicious behavior, to pinpoint common causes, and to identify the extent to which other malicious activities have occurred.

Figure 6.6 shows an overview of the PICO SDN architecture. PICO SDN has two phases: a *run-time phase* (Section 6.5.1) that collects relevant provenance information during execution, and an

investigation phase (Section 6.5.2) that analyzes the provenance.

PicoSDN is designed with the following goals in mind:

- G1** *Precise Dependencies.* PicoSDN should reduce the units of execution to remove false execution dependencies that arise from long-running processes in the SDN control plane. PicoSDN should also reduce the unit size of data to remove false data dependencies.
- G2** *Unified Network Model.* PicoSDN should leverage control *and* data plane activities, and thereby mitigate the incomplete dependency problem.
- G3** *Iterative Analysis.* PicoSDN should perform backward and forward tracing to enable causal analysis of SDN attacks. It should efficiently summarize network activities and network identifier evolution.
- G4** *Activity Completeness.* PicoSDN should observe and record any apps, controller, or data plane activity relevant to network activities to ensure that it serves as a control plane reference monitor.

6.5.1 Runtime Phase

During the network's execution, PicoSDN's runtime phase records control plane activities in its *collector* and transforms them into a lightweight graph by using its *serializer*.

Collector The provenance collector consists of three components: *wrappers* around event dispatches and packet processors, *hooks* on API calls, and an *internal state* tracker.

We have instrumented wrappers around the SDN controller's event dispatcher and packet processor. The provenance collector uses these wrappers to maintain knowledge about which event listener or packet processor is currently handling the dispatch or processing, respectively; this achieves goal **G1**.

We have instrumented hooks on each of the SDN controller's API calls; this achieves goal **G4**. For a single-threaded controller, the reconstruction of the sequence of events, packets, and API calls is straightforward. However, in modern multi-threaded controllers, we also need a concurrency model to correctly link such calls to the right events. For event dispatching, we assume the following concurrency model: a particular event, ϵ_1 , is processed *sequentially* by each interested event listener (*i.e.*, ϵ_1 is processed by listener l_1 , then by l_2); different events, ϵ_1 and ϵ_2 , may be processed *concurrently* (*i.e.*, ϵ_1 is processed by listener l_1 followed by l_2 , while concurrently ϵ_2 is processed by listener l_3 followed by l_4). That is the model used by ONOS,⁴ among other SDN controllers. It allows

⁴ONOS maintains several event dispatch queues based on the event type, and each queue is implemented in a sepa-

PiCoSDN’s provenance collector to use hooks to correctly determine whether a particular API call should link the use or generation of control plane objects to the event listener (or packet processor) in execution at that time. Hooking the API calls and linking them with the event and packet wrappers not only permits a transparent interposition over all app and data plane interactions with the control plane, but also avoids the limitations of prior work [129] that requires app instrumentation.

The provenance collector includes an internal state tracker that maintains knowledge of current events and control plane objects to detect when such objects change. The internal state is necessary to keep track of ephemeral objects’ uniqueness that would not necessarily be captured by raw logging alone. (See Section 6.8 for a discussion about internal state storage costs and external provenance storage costs.)

Serializer Once the provenance collector has determined the correct provenance based on context, the provenance serializer writes out a lightweight serialized graph of nodes and edges.

6.5.2 Investigation Phase

At some later point in time, PiCoSDN’s investigation phase uses the lightweight serialized graph as a basis for analysis. The *ingester* de-serializes the graph, the *cleaner* removes unnecessary provenance, and the *topology augments* incorporates the data plane model. The *tracer* answers practitioner queries. Each component is designed to be modular.

6.5.2.1 Ingestor, cleaner, and topology augments

The ingester reads in the serialized graph. As most nodes contain additional details, the graph ingester de-serializes the node’s dictionary into a set of key–value pairs. The cleaner component can perform preprocessing to remove unnecessary or irrelevant nodes and edges. For instance, the cleaner removes singleton nodes that are not connected to anything; they may appear if they are not being used. The cleaner removes nodes that are not relevant to an investigation; for instance, removing Statistic nodes about traffic counts may be useful if the investigation does not involve traffic counts. The topology augments adds edges into the graph (*e.g.*, *wasDerivedFrom* relations between PacketIns and PacketOuts) to define the data plane model; doing so achieves goal **G2**.

PiCoSDN’s data plane model algorithm is shown in Algorithm 6.1. We assume that the data plane’s topology can vary over time, and for each variation, we say that the state is an *epoch* consisting of a topology that is valid between a start time and an end time (lines 1–2). For each PacketIn, we want

rate thread. Given that listeners process a particular event sequentially, ONOS’s event dispatcher sets a hard time limit for each event listener to avoid indefinite halting.

Algorithm 6.1 Data Plane Model

Input: graph \mathcal{G} , data plane topology states \mathcal{D}_{set} , time window τ_w , headers fields to match on H
Output: graph with data plane model \mathcal{G}
Initialize: $(\mathcal{V}, \mathcal{E}) \leftarrow \mathcal{G}$

```
1: for each  $\mathcal{D} \in \mathcal{D}_{set}$  do
2:    $(\mathcal{N}, \tau_{start}, \tau_{end}) \leftarrow \mathcal{D}$   $\triangleright$  Data plane topology graph  $\mathcal{N}$ , epoch start  $\tau_{start}$ , epoch end  $\tau_{end}$ 
3:    $(\mathcal{N}_{switches}, \mathcal{N}_{links}) \leftarrow \mathcal{N}$ 
4:   for each  $p_{in} \in \mathcal{V}_{class=PacketIn}$  do  $\triangleright$  Packet  $p_{in}$ 
5:     if  $\tau_{start} < p_{in}.ts < \tau_{end}$  then  $\triangleright$  Timestamp  $p_{in}.ts$ 
6:       for each  $p_{out} \in \mathcal{V}_{class=PacketOut}$  do
7:         if  $(p_{out}.switch, p_{in}.switch) \in \mathcal{N}_{links}$  then
8:           if  $p_{out}.H = p_{in}.H$  then
9:             if  $p_{out}.ts < p_{in}.ts$  and  $p_{in}.ts - p_{out}.ts \leq \tau_w$  then
10:               $\mathcal{V} \leftarrow \mathcal{V} \cup \{(p_{in}, p_{out})\}$ 
11:            end if
12:          end if
13:        end if
14:      end for
15:    end if
16:  end for
17: end for
18:  $\mathcal{G} \leftarrow (\mathcal{V}, \mathcal{E})$ 
19: return  $\mathcal{G}$ 
```

to determine if it should link to a causally related PacketOut (line 4). PICO SDN filters temporally based on the current epoch (line 5), and it checks all PacketOuts during that epoch (line 6). We consider a PacketOut to be causally related to a PacketIn if all of the following conditions are met: 1) there is a link between the outgoing and incoming switches (line 7); 2) the specified packet headers are the same for both packets (line 8); 3) the PacketOut “happened before” the PacketIn (line 9); and 4) the timestamp differences between the PacketOut and PacketIn are within a specific threshold (line 9).

As PICO SDN is modular, Algorithm 6.1’s data plane model can be replaced as needed. For instance, header space analysis [103] uses functional transformations to model how packets are transformed across the data plane (e.g., packet modifications), and P4 [36] proposes a programmable data plane. Practitioners can write their own data plane model components that take those transformations into account.

6.5.2.2 Tracer

After the graph is prepared, the tracer component answers investigative queries. PICO SDN provides facilities to answer queries related to root cause analysis, network activity summarization, and net-

Algorithm 6.2 Common Ancestry Trace

Input: graph \mathcal{G} , evidence set N
Output: agent set Ag , activity set Ac , and entity set En
Initialize: $(\mathcal{V}, \mathcal{E}) \leftarrow \mathcal{G}$, $Ag \leftarrow \emptyset$, $Ac \leftarrow \emptyset$, $En \leftarrow \emptyset$, $A \leftarrow \mathcal{V}$

```
1: for each  $e \in \mathcal{E}$  do                                ▷ Remove non-causal edges
2:   if  $e$  is a wasRevisionOf edge then
3:      $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$ 
4:   end if
5: end for
6: for each  $n \in N$  do                                ▷ Evidence  $n$  (note:  $n \in \mathcal{V}$ ,  $N \subset \mathcal{V}$ )
7:    $A_n \leftarrow \text{getAncestors}((\mathcal{V}, \mathcal{E}), n)$         ▷ Set of ancestor nodes  $A_n$ 
8:    $A \leftarrow A \cap A_n$                             ▷ Common ancestor set  $A$ 
9: end for
10: for each  $a \in A$  do                                ▷ Common ancestor  $a$ 
11:   if  $a$  is an Agent node then
12:      $Ag \leftarrow Ag \cup a$ 
13:   else if  $a$  is an Activity node then
14:      $Ac \leftarrow Ac \cup a$ 
15:   else
16:      $En \leftarrow En \cup a$ 
17:   end if
18: end for
19: return  $(Ag, Ac, En)$                                 ▷  $Ag \subset \mathcal{V}$ ,  $Ac \subset \mathcal{V}$ ,  $En \subset \mathcal{V}$ 
```

work state evolution; these facilities achieve goal **G3**. We now describe each kind of query and under what scenarios a practitioner would want to use each kind.

As \mathcal{G} is a DAG, we assume the use of standard graph functions in Algorithms 6.2–6.5 that can determine the ancestor and descendant nodes (*i.e.*, progeny) of a given node n , denoted by $\text{getAncestors}(\mathcal{G}, n)$ and $\text{getDescendants}(\mathcal{G}, n)$, respectively.

Root cause analysis After an attack, a practitioner wishes to investigate the attack’s causes so as to determine what changes should be made to prevent such attacks from reoccurring. We assume that a practitioner has *evidence* of incorrect behavior, wants to find common causes, and wants to explore whether other evidence of incorrect behavior also exists. PICO SDN provides two interrelated algorithms to achieve these goals: *common ancestry tracing* (Algorithm 6.2) and *backward-forward tracing* (Algorithm 6.3). Practitioners can iteratively use these tools to determine root causes efficiently.

Algorithm 6.2 shows the common ancestry tracing. We assume that our practitioner can pinpoint evidence of incorrect behavior, such as a set of packets or flow rules that appear suspicious. Our practitioner’s goal is to see if such evidence has anything in common with past history. PICO SDN starts by discarding non-causal edges in the graph (lines 1–3). Then, for each piece of evidence,

Algorithm 6.3 Iterative Backward-Forward Trace

Input: graph \mathcal{G} , evidence n , root r
Output: affected difference function $\Delta : \mathcal{V} \rightarrow \mathfrak{P}(\mathcal{V})$
Initialize: $(\mathcal{V}, \mathcal{E}) \leftarrow \mathcal{G}; \Delta(i) \leftarrow \emptyset, \forall i \in \mathcal{V}$

```
1: for each  $e \in \mathcal{E}$  do                                ▷ Remove non-causal edges
2:   if  $e$  is a wasRevisionOf edge then
3:      $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$ 
4:   end if
5: end for
6:  $A_n \leftarrow \text{getAncestors}((\mathcal{V}, \mathcal{E}), n)$           ▷ Evidence's ancestor set  $A_n$ 
7:  $D_r \leftarrow \text{getDescendants}((\mathcal{V}, \mathcal{E}), r)$           ▷ Root's descendant set  $D_r$ 
8:  $\mathcal{V}_{\text{intermediate}} \leftarrow A_n \cap D_r$ 
9: for each  $v_i \in \mathcal{V}_{\text{intermediate}}$  do
10:   $\Delta(i) \leftarrow D_r \setminus \text{getDescendants}((\mathcal{V}, \mathcal{E}), v_i)$ 
11: end for
12: return  $(\mathcal{V}_{\text{intermediate}}, \Delta)$ 
```

PICO SDN computes its set of ancestor nodes and takes the intersection of that ancestry with the ancestries of all previous pieces of evidence (lines 4–6). Once all the pieces of evidence have been examined, the set of common ancestors is partitioned into agent, activity, and entity nodes (lines 7–13). Thus, PICO SDN provides data-centric, process-centric, and agent-centric answers.

Algorithm 6.3 shows the iterative backward-forward tracing. Our practitioner has a piece of evidence and a suspected root cause (derived, perhaps, from Algorithm 6.2). Our practitioner's goal is to iteratively determine how intermediate causes (*i.e.*, those causes that lie temporally in between the evidence and the root cause) impact the evidence and other effects on the network's state. PICO SDN starts by discarding non-causal edges in the graph (lines 1–3). For the piece of evidence, PICO SDN determines all of its ancestors, or the set of all causally related entities, activities, and agents responsible for the evidence (line 4). For the suspected root cause, PICO SDN determines all of its descendants, or the set of all the entities and activities that the root cause affected (line 5). PICO SDN takes the intersection of those two sets (line 6) to examine only the intermediate causes that occurred as a result of the root cause. For each intermediate cause, PICO SDN derives the set of affected entities and activities that the root cause affected that the intermediate cause did not affect (lines 7–8). In essence, that lets the practitioner iteratively examine intermediate effects at each stage.

Network activity summarization One general provenance challenge is that graphs can become large and difficult to interpret even for simple activities, and that creates fatigue when one is analyzing such graphs for threats and attacks [240]. PICO SDN provides an efficient network-specific summarization.

Algorithm 6.4 Network Activity Summarization

Input: graph \mathcal{G}
Output: set of (activity a , flow rule f_{out} , packet p_{in} , data plane packets P_{in})
Initialize: $(\mathcal{V}, \mathcal{E}) \leftarrow \mathcal{G}$, $S \leftarrow \emptyset$

```
1: for each  $e \in \mathcal{E}$  do                                     ▷ Remove non-causal edges
2:   if  $e$  is a wasRevisionOf edge then
3:      $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$ 
4:   end if
5: end for
6: for each  $a \in \mathcal{V}_{\text{class=Activity}}$  do
7:    $f_{out} \leftarrow \text{null}$ ,  $p_{in} \leftarrow \text{null}$ ,  $P_{in} \leftarrow \text{null}$ 
8:    $P_{in} \leftarrow \text{getAncestors}((\mathcal{V}, \mathcal{E}), a)$ 
9:   for each  $p \in P_{in}$  do
10:    if  $p \notin \mathcal{V}_{\text{class=PacketIn}}$  then
11:       $P_{in} \leftarrow P_{in} \setminus \{p\}$ 
12:    end if
13:  end for
14:  if  $\langle a \rightarrow (v \in \mathcal{V}_{\text{class}\neq\text{Activity}} \text{ or } e \in \mathcal{E})^* \rightarrow p \in \mathcal{V}_{\text{class=PacketIn}} \rangle$  backward trace path exists then
15:     $p_{in} \leftarrow p$ 
16:  end if
17:  if  $\langle f \in \mathcal{V}_{\text{class=FlowRule}} \rightarrow (v \in \mathcal{V}_{\text{class}\neq\text{Activity}} \text{ or } e \in \mathcal{E})^* \rightarrow a \rangle$  backward trace path exists then
18:     $f_{out} \leftarrow f$ 
19:  end if
20:   $S \leftarrow S \cup \{(a, f_{out}, p_{in}, P_{in})\}$ 
21: end for
22: return  $S$ 
```

Algorithm 6.4 shows the summarization approach. Our practitioner’s goal is to answer questions of the form “Which data plane activities (*i.e.*, packets) caused flow rules to be or not be installed?” PICO SDN starts by discarding non-causal edges in the graph (lines 1–3). It collects each event listener or packet processor activity (line 4). For each activity, it derives all of the PacketIn packets that causally affected the activity (lines 5–9). Then, PICO SDN determines whether a PacketIn is a direct⁵ cause by computing a backward trace path; if it is a direct cause, the packet is marked (lines 10–11). Similarly, PICO SDN determines whether a FlowRule is a direct effect of the activity; if it is, the flow rule is marked (lines 12–13).

Algorithm 6.4 allows practitioners to efficiently investigate instances in which flow rules were *not* created, too. For example, if an event listener used a packet but did not generate a flow rule, the resulting value for f_{out} would be null. Algorithm 6.4 also derives a set of all data plane PacketIn packets causally related to each activity; as we show later in Section 6.7, this information is useful for diagnosing cross-plane attacks.

⁵In other words, without any intermediate Activity nodes in between. However, intermediate data derivations between Entity objects are permissible.

Algorithm 6.5 Network Identifier Evolution

Input: graph \mathcal{G} , network identifier i
Output: revision trace path t_r , affected nodes function F
Initialize: $(\mathcal{V}, \mathcal{E}) \leftarrow \mathcal{G}$; $\mathcal{E}_{stash} \leftarrow \emptyset$; $F(i) \leftarrow \emptyset, \forall i \in \mathcal{V}$

```
1: for each  $e \in \mathcal{E}$  do                                     ▷ Remove and stash non-causal edges
2:   if  $e$  is a wasRevisionOf edge then
3:      $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$ 
4:      $\mathcal{E}_{stash} \leftarrow \mathcal{E}_{stash} \cup \{e\}$ 
5:   end if
6: end for
7:  $n \leftarrow \text{getMostRecentNode}(\mathcal{V}, i)$ 
8:  $t_r \leftarrow \langle n \rangle$ 
9:  $F(n) \leftarrow \text{getDescendants}((\mathcal{V}, \mathcal{E}), n)$ 
10: while  $n \leftarrow \text{getNextNode}(\mathcal{E}_{stash})$  and  $n$  is not null do
11:    $t_r.\text{append}(\text{wasRevisionOf}, n)$ 
12:    $F(n) \leftarrow \text{getDescendants}((\mathcal{V}, \mathcal{E}), n)$ 
13: end while
14: return  $(t_r, F)$ 
```

Network state evolution Given the attribution challenges of data plane host activities, practitioners will want to investigate whether any of the pertinent identifiers have been spoofed. Such spoofing can have systemic consequences for subsequent control plane decisions [22, 44, 55, 93]. PICO SDN efficiently tracks network identifier evolution (*i.e.*, the wasRevisionOf relation) and provides an algorithm to query it (Algorithm 6.5).

Algorithm 6.5 shows the network identifier evolution approach. Our practitioner’s goal is to see whether any identifiers have evolved over time as a result of malicious spoofing, as well as the extent of damage that such spoofing has caused. PICO SDN starts by stashing non-causal edges in the graph, thus removing them from causality-related processing, but keeping them for reference (lines 1–4). For a given network identifier, PICO SDN determines the node most recently linked to that identifier (line 5) and adds it to a revision trace path (line 6). PICO SDN derives that node’s descendants to determine the extent to which that network identifier causally affected other parts of the network state (line 7). That process is repeated back to the identifier’s first version (lines 8–10).

Algorithm 6.5 produces a concise representation of an identifier’s state changes over time. That allows the practitioner to easily determine when an identifier may have been spoofed, and that respective node in time can be used in Algorithm 6.3 as a root cause in further iterative root cause analysis. Furthermore, the affected nodes that are returned by Algorithm 6.5 can be used as evidence in the common ancestry trace of Algorithm 6.2.

Table 6.3: List of PicoSDN Hooks (*i.e.*, PicoSDN API Calls).

PicoSDN API call	Description
<code>recordDispatch(<i>activity</i>)</code>	Mark the start of an event dispatch or packet processing loop
<code>recordListen(<i>activity</i>)</code>	Mark the demarcation (<i>i.e.</i> , start of each loop) of an event being listened to or a packet being processed
<code>recordApiCall(<i>type</i>, <i>entity</i>)</code>	Record a control plane API call of a <i>type</i> (<i>i.e.</i> , create, read, update, delete) on an <i>entity</i> (or <i>entities</i>)
<code>recordDerivation(<i>entity</i>, <i>entity</i>)</code>	Record an <i>object</i> derived from another <i>object</i>

6.6 Implementation

We implemented PicoSDN in Java on ONOS v1.14.0 [26]. We modified ONOS in several key locations. We created a set of PicoSDN API calls, which are listed in Table 6.3. We created Java classes to represent Activity and Entity objects, and we made them into superclasses for relevant ONOS classes (*e.g.*, ONOS’s Packet superclass is Entity). We wrapped the ONOS event dispatcher and packet processor by using the `recordDispatch()` and `recordListen()` calls, which represented the execution partitioning of PicoSDN. We hooked the ONOS core services’⁶ public API calls by using the `recordApiCall()` calls.⁷ For a given core service API call, if the return value was iterable, we marked each object within the iterable object with its own separate provenance record. For certain data whose processing spanned multiple threads, we used `recordDerivation()` calls to maintain the causal relations across threads. We implemented the ingester, modifier, and tracer on top of the JGraphT library [190].

Because of our design decisions, described in Section 6.5.1, we did not need to perform an analysis on or make any modifications to the ONOS apps. Practitioners do not need to instrument each new app that they install in their network. Furthermore, PicoSDN’s API and classes allow PicoSDN to be easily updated as new core services and objects are implemented in ONOS. Although we implemented PicoSDN on ONOS, the same conceptual provenance model and design can be implemented with minimal modifications on any event-based SDN controller architecture, and indeed the most popular controllers (*e.g.*, ODL [30] and Floodlight [28]) all use such architectures.

⁶In ONOS, these core services are represented by classes that end in `*Manager` or `*Provider`. For instance, ONOS has a `HostManager` class and a `HostProvider` class that include public API calls related to hosts.

⁷As ONOS does not provide a reference monitor architecture that would allow us to wrap one central interposition point across *all* API calls, we had to add `recordApiCall()` hooks across 141 API calls to ensure completeness.

6.7 Evaluation

We now evaluate PICO SDN’s performance and analysis capabilities. We have examined its performance overhead (Section 6.7.1). We used recent SDN attacks to show that PICO SDN can capture a broad diversity of SDN attacks. We consider the following cases for our security analysis evaluation: the earlier motivating example [44] (§6.7.2), a cross-plane host migration attack [22] (Section 6.7.3), and a cross-app attack (Section 6.7.4). We implemented all topologies using Mininet [41].⁸ We ran experiments using a workstation with a four-core 3.30-GHz Intel Core i5-4590 processor and 16 GB of memory.

6.7.1 Performance Evaluation

Given the latency-critical nature of control plane decision-making, we benchmarked the latency that PICO SDN imposed on common ONOS API calls (Figure 6.7a). To further understand these costs, we microbenchmarked PICO SDN’s hooks (Figure 6.7b). We also benchmarked the overall latency imposed by a reactive control plane configuration (Figure 6.7c) as a function of the data plane’s network diameter.

Benchmarks on ONOS Figure 6.7a shows the average latencies of common ONOS API calls with and without PICO SDN enabled. These calls were called most often in our case studies (Sections 6.7.2, 6.7.3, and 6.7.4) and relate to flow rules, hosts, and packets. Although certain calls generated significantly greater latency, that was expected for cases in which iterable objects require generation of individual provenance records.

Microbenchmarks To further analyze the benchmark results, we microbenchmarked PICO SDN’s hooks (*i.e.*, PICO SDN’s API calls). Figure 6.7b shows the average latencies of the PICO SDN API calls listed in Table 6.3, with the `recordApiCall()` calls broken down by call type. As shown in Figure 6.7b, event listening and dispatching are fast operations. We expected API calls to be slower, given the tracking operations within PICO SDN’s internal state.

Overall latency We also measured the overall latency that PICO SDN imposes on control plane operations. We wanted to see what the additional incurred latency would be from the perspective of host-to-host communication, or the *time-to-first-byte* metric. This metric measures the total round-trip time (RTT) measured between data plane hosts (*e.g.*, via the `ping` utility) for the first packet

⁸We chose Mininet because it is common in prior work (*e.g.*, [51, 129]) and because it causes PICO SDN’s runtime phase to record the same kind and amount of provenance information that would be captured in a real network.

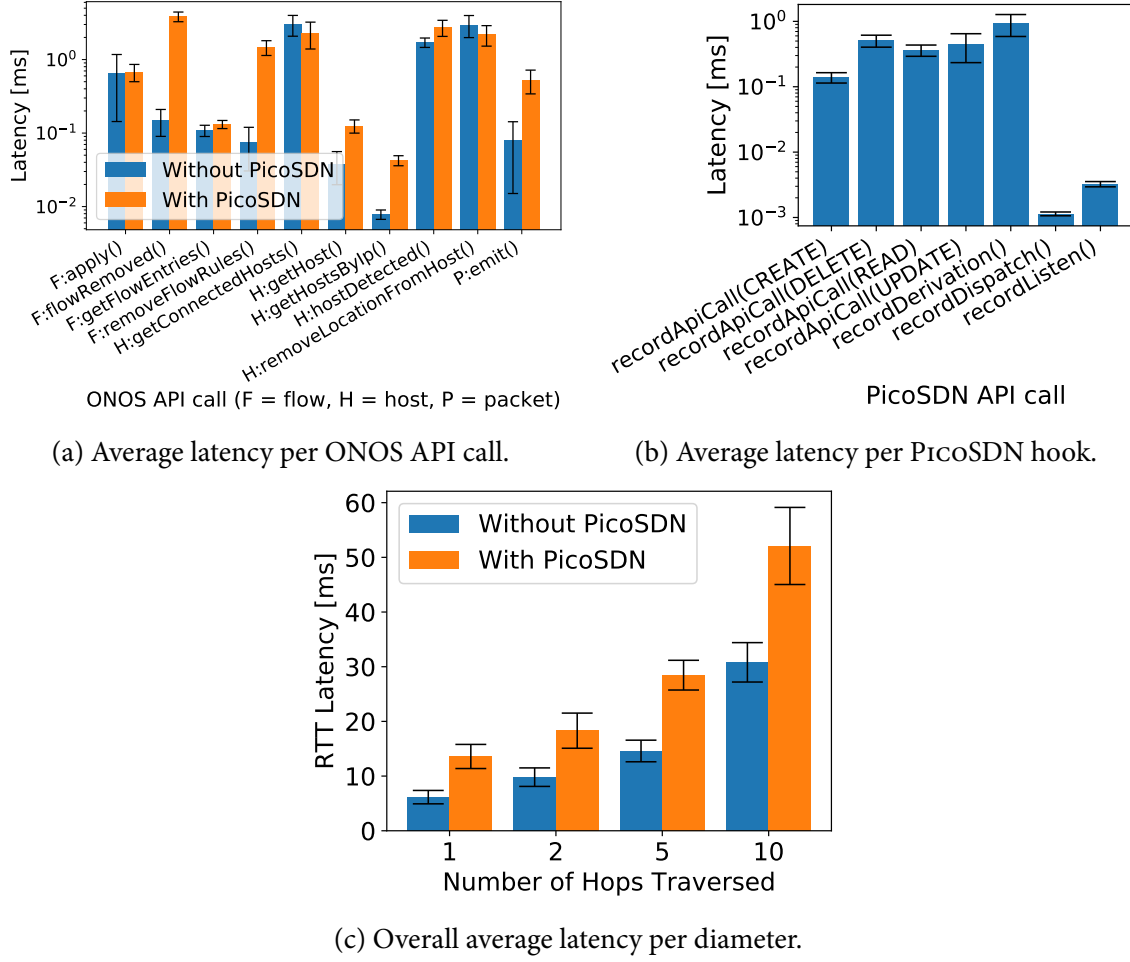


Figure 6.7: PicoSDN latency performance results. (Error bars represent 95% confidence intervals.)

of a flow. The RTT captures the latency of both data plane processing and control plane decision-making.

In *reactive* control planes, the first packet of a flow suffers high latency because it does not match existing flow rules, but once matching flow rules have been installed, the remaining packets of the flow use the data plane’s fast path. Although SDN configurations can be *proactive* by installing flow rules before any packets match them, we measured a reactive configuration because it represents the *worst-case* latency that is imposed if the controller must make a decision at the time it sees the first packet. (See Section 6.8 for a discussion of the differences.) In addition, the network’s diameter (*i.e.*, the number of hops between data plane hosts) affects latency in reactive configurations if the first packet must be sent to the controller *at each hop*. Thus, we measured a reactive configuration and varied the number of hops to determine the effect on latency.

Figure 6.7c shows the average overall latencies imposed with and without PicoSDN, varied by the number of hops. We performed each experiment over 30 trials. In contrast to prior work [51, 129],

we parameterized the number of hops traversed to reflect different network topology diameters. We found that PICO SDN increased the overall latency on average from 7.44 ms for 1-hop (*i.e.*, same-switch) topologies to 21.3 ms for 10-hop topologies. That increase was expected, given that additional provenance must be generated for longer routes. For long-running flow rules, the one-time latency cost in the flow’s first packet can be amortized. Thus, we find PICO SDN acceptable for practical implementation.

6.7.2 Security Analysis 1: Motivating Example

We now revisit the motivating cross-plane attack example described in Section 6.2.1. Our practitioner now examines the provenance data collected during the attack by PICO SDN’s runtime phase, which is shown in abbreviated form in Figure 6.2b.

As our practitioner knows that hosts h_1 and h_2 communicated, he or she uses the network activity summarization to derive the set of flow rules related to these hosts. Among the returned set, the practitioner sees the following:

1. the flow rule from the fwd app that allowed communication $(\text{fwd}, f_3, p_4, \{p_3, p_2\})$;
2. the acl app’s failure to install a flow denial rule, resulting from an invalid IP address $(\text{acl}, \text{null}, \text{null}, \{p_1\})$; and
3. the acl app’s failure to install a flow denial rule, resulting from the host event type’s not being handled $(\text{acl}, \text{null}, \text{null}, \{p_2\})$.

The practitioner uses the common ancestry trace of fwd and acl’s actions to determine the common ancestors of the discovered flow rules. Among this set, the common ancestor is the switch port agent s_1 : port 1. Now equipped with a set of possible root causes, the practitioner issues a backward-forward trace from f_3 to the root of the switch port agent to see the differences in descendants (*i.e.*, impacts) that each intermediate cause affects. That allows the practitioner to discover that the relevant root cause can be traced back to the spoofed packet p_1 . Starting there, the practitioner’s forward traces show the effects that p_1 has on the network’s subsequent activities, such as the corrupted host object $h_{1(v1)}$. PICO SDN identifies the root cause and effects of the spoofed packet, thus letting the practitioner know that host h_1 should be disconnected.

Figure 6.2b shows the relevant features that PICO SDN produced. The data plane model clearly links the data plane packets that result from fwd’s installation across switches, which prior tools do not. The default flow rules would otherwise create a dependency explosion, but PICO SDN mitigates that problem by partitioning agency with switch ports.

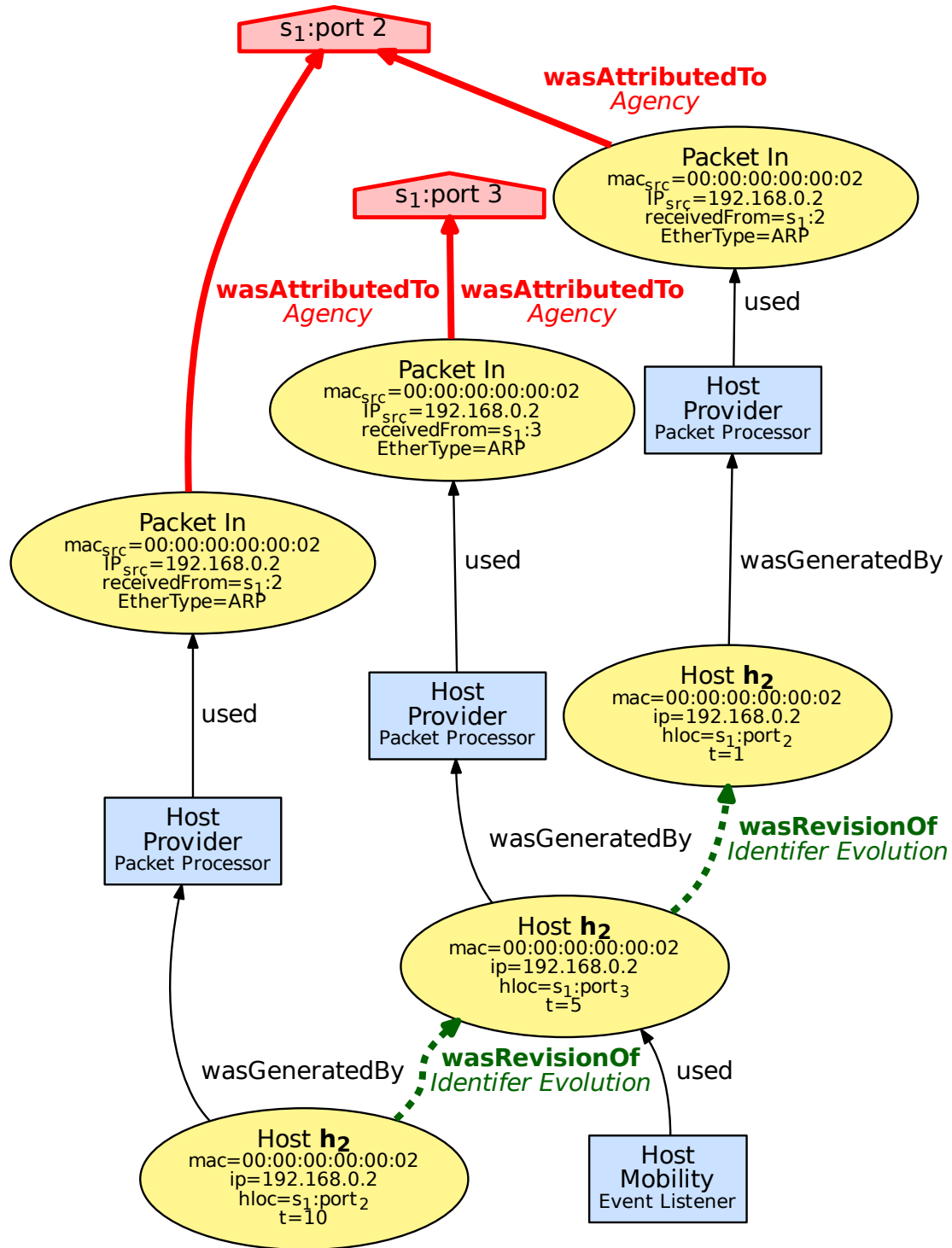


Figure 6.8: Relevant features of the host migration attack's graph showing the evolution of hosts that claimed to be h_2 .

6.7.3 Security Analysis 2: Host Migration

We consider another cross-plane-based host migration attack. This attack uses a malicious data plane host to trick the control plane into believing that a victim host has changed its location. We assume a three host (h_1 , h_2 , and h_3) topology with one switch (s_1). Host h_3 attempts to masquerade as host h_2 so as to trick other hosts (e.g., h_1) into sending traffic meant for h_2 to instead go to h_3 . (See [22] for the attack's details.)

Our practitioner queries the network identifier evolution for h_2 . Figure 6.8 shows a partial provenance graph of the relevant features. The evolution shows that h_2 appears to have switched network ports from s_1 's port 2 to port 3; in reality, h_3 spoofed h_2 's identifier. The query returns the descendants (*i.e.*, the impacts) that each version of the identifier has had on the network. For instance, during the time that the spoofed location of h_2 was being used between times $t = [5, 10]$, old flow rules that directed traffic to h_2 were removed by the host mobility app. The practitioner can now efficiently see the attack's ramifications at each stage because of the combination of the network identifier evolution and the forward-tracing capabilities, which prior work does not offer. PICO SDN identifies a cause in the spoofed packet used by the host provider, and also finds the other effects of the spoofed packet. The practitioner thus disconnects the malicious host from port 3.

6.7.4 Security Analysis 3: Cross-App Attack

We also use PICO SDN to analyze a cross-app-based attack. This attack uses a malicious app to modify packets in the packet processing pipeline, which subsequent apps use to make control plane decisions. (We refer the reader to [51] for a detailed description of the attack's mechanism.)

Figure 6.9 shows the important features of the graph. We can see that the packet changes as it is handed off from the triggering trigger (*i.e.*, malicious) app to the forwarding fwd (*i.e.*, benign) app in the processing pipeline. Since PICO SDN uses an event-based model, we can reduce the false dependencies that PROV SDN would show. For instance, for each instance of trigger's event handler, the precise API calls that were used are embedded in the used and wasGeneratedBy relations for API read and write calls, respectively, on the PacketIns.

To understand how the attack occurred, a practitioner issues a network activity summarization query to find malicious flow rules and uses them in the common ancestry trace to look at the trigger agent. The practitioner then issues an iterative backward-forward trace query on the trigger app to determine the extent to which trigger has caused other undesired network activities. PICO SDN identifies the root cause and other effects of trigger, thus informing the practitioner that the app should be removed.

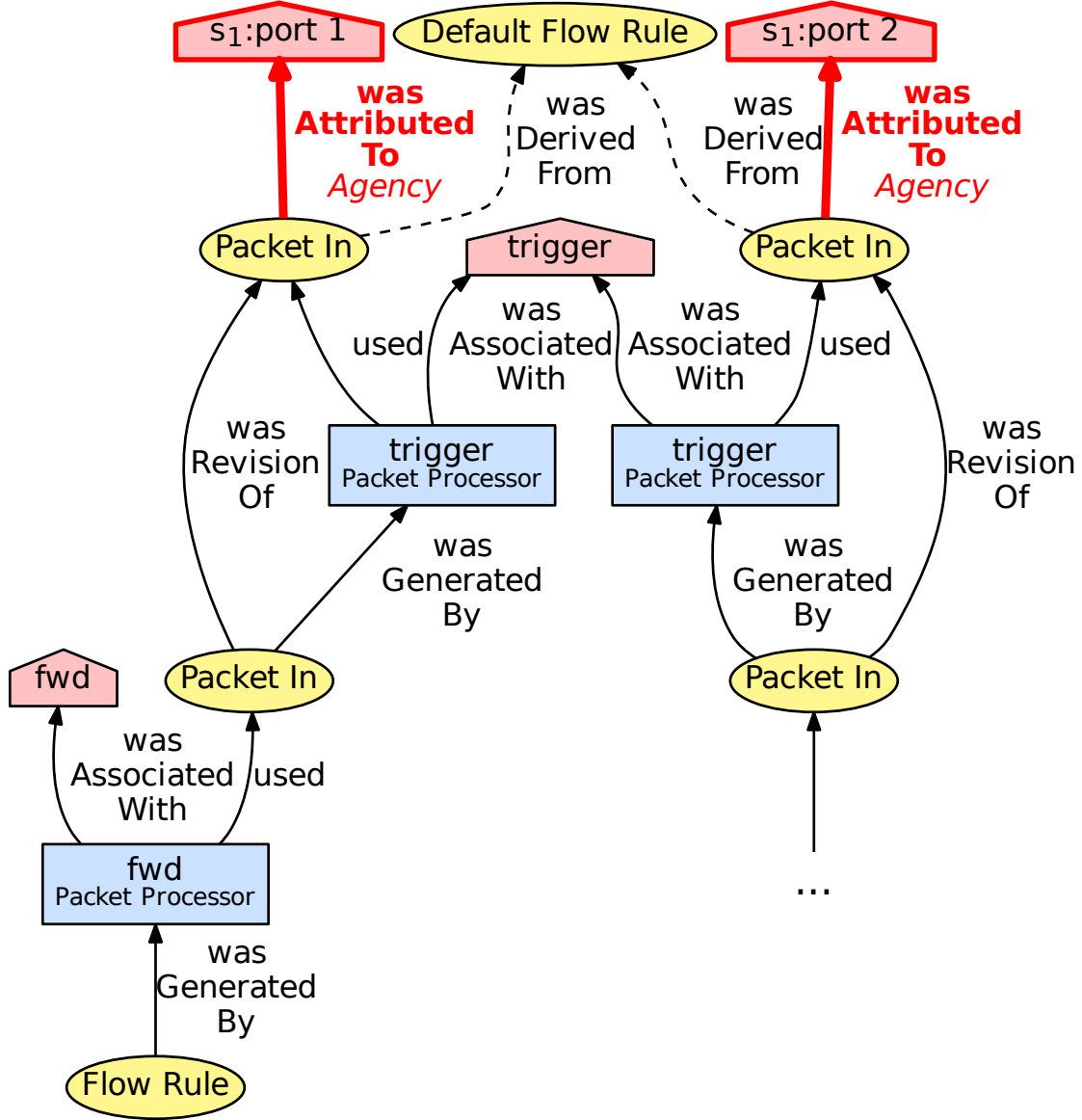


Figure 6.9: Relevant features of the graph from the cross-app attack. The graph shows that trigger modifies packets before to fwd receives them.

6.8 Discussion

Reactive and proactive configurations PicoSDN is designed to work for both reactive and proactive SDN control plane configurations. We used reactive configurations in our case studies because recent SDN attacks have leveraged reactive configurations [22, 43, 44, 55], but we argue that PicoSDN is well-suited for proactive configurations, too. Proactive configurations install flow rules ahead of time. However, the time at which flow rules are inserted may be far removed from the time when data plane packets exercise these rules. As a result of the time gap, manual tracing by a practitioner would be a difficult task. That provides the motivation to create quality network forensics

tools such as PICO SDN to maintain history.

Storage costs Although a reduction of PICO SDN’s latency overhead will be critical for practical implementation, the storage overhead must also be a consideration. Internally, PICO SDN’s design allows it to maintain only the minimum state necessary to keep track of object changes. In practice, that means that the state is as large as the number of objects representing the network’s flow rules, topology, and system principals (*e.g.*, switches and hosts) at a given time. Externally, provenance graphs can grow large over time if they must maintain historical metadata. While provenance storage reduction is an important goal, it is orthogonal to the aims of PICO SDN and can be implemented using existing provenance storage reduction systems and techniques [241, 242, 243, 244, 245, 246, 247].

Attack identification PICO SDN does not claim to automatically identify and detect SDN attacks. SDN attack detection is an ongoing research area examining expected semantic behavior [22, 55, 93] and pattern recognition of anomalous features or behavior [44, 114], both of which are orthogonal to PICO SDN’s aims. PICO SDN provides practitioners with the necessary tools for *insight* into control plane execution and *analysis* of causal dependencies. Both are necessary for successful attack identification.

6.9 Related Work

SDN control plane insight FORENGUARD [129] is the prior effort that is most closely related to PICO SDN. Like FORENGUARD, PICO SDN provides root cause analysis capabilities for SDN attacks. PICO SDN extends those capabilities with a data plane model and mitigates the data dependency explosions caused by default flow rules. PROVSDN [51] focuses on information flow control enforcement rather than root cause analysis, so its analysis capabilities are limited; it also uses an API-centric model rather than an event-centric model for execution partitioning, resulting in false dependencies that would not be generated in PICO SDN’s provenance model. GitFlow [141] proposes a version control system for SDN; that influenced our decision to include revision relations. AIM-SDN [248] outlines the challenges in SDN, influencing our decisions on how to represent agency. Ujich *et al.* [122] argue that provenance is necessary to ensure a secure SDN architecture.

Declarative network provenance has shown promise in automated bug removal [249], differential provenance [202, 250], meta provenance [132], and negative provenance [130, 131]. The various solutions use a declarative paradigm [251], which requires nontrivial translation for apps written in the imperative paradigm. A benefit of declarative programs is that they inherently capture the data plane model, which PICO SDN provides but PROVSDN and FORENGUARD do not.

The general research space of SDN security, including the set of potential attack vectors, is large and well-studied; we refer the reader to [80] for a survey of the area.

SDN debugging and verification We outline existing SDN debugging and verification tools, as they are complementary to provenance-based causal analysis tools.

Control-plane debugging tools include FALCON [128], Net2Text [127], BigBug [126], ConGuard [109], STS [110], and OFRewind [123]. They record the network’s state to identify unusual behavior and replay suspect activities in a simulated environment. However, they assume that activity traces are dependent upon all previous states and/or inputs, whereas PICO SDN avoids that assumption through its dependency partitioning.

Data plane verification tools include Cocoon [121], SDNRacer [107], VeriCon [106], VeriFlow [96], NetPlumber [97], NICE [105], NDB [125], header space analysis [103], and Anteater [104]. They prevent instantiation of incorrect configurations in the network according to a predefined policy, but such tools’ prevention capabilities are dependent upon correct policy specifications. PICO SDN records known and unknown attacks so that practitioners can investigate how such attacks occurred.

Provenance and causality analysis The dependency explosion problem has been studied for host applications [183], binary analysis [182, 252], and host operating systems [178, 240, 253, 254, 255, 256, 257]. Provenance for attack causality analysis has also been well-studied [3, 176, 179, 187, 239, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270]. PICO SDN’s primary contributions to this area include 1) a provenance model for SDN control and data planes that focuses on SDN-specific dependency explosion factors (*e.g.*, default flow rule dependencies), and 2) relevant attack analysis techniques of particular interest to network practitioners (*e.g.*, network summarization).

6.10 Conclusion

We presented PICO SDN, a provenance-informed causal observation tool for SDN attacks. PICO SDN leverages a fine-grained provenance model to allow practitioners to reconstruct past control and data plane activities, to analyze them for root causes when control plane attacks occur, to understand the scope of attacks’ effects on other network activities, and to succinctly summarize the network’s activities and evolution. We evaluated PICO SDN using recent control plane attacks, and we found that PICO SDN is practical for runtime collection and offline analysis.

CHAPTER 7

CONCLUSIONS

Programmable networks that use the SDN architecture have enabled unmatched flexibility in the enforcement of network security and policy. However, the security posture of the SDN architecture is significantly different from that of traditional networking architectures because of that programmable design. The “appification” of network functionalities and the coordination of those functionalities within a network operating system creates new security challenges.

In the work described in this dissertation, we demonstrated that the security posture of SDN can be enhanced by using control and data dependency techniques that track information flow and enable understanding of application composability, control and data plane decoupling, and control plane insight. We applied data provenance and static analysis techniques to the SDN architecture, which allowed us to better understand and analyze the architecture’s security posture. Our focus was on development of attacks that revealed cross-app and cross-plane vulnerabilities, as well as on design of defenses to prevent or mitigate such attacks. We provided contributions for the runtime, pre-runtime, and post-runtime stages of attack prevention and mitigation.

In this chapter, we briefly review the significant contributions of this dissertation, provide overall takeaways based on those contributions, and outline future research directions that would extend the work of this dissertation.

7.1 Review of Contributions

We applied a conceptual framework of accountability to the SDN architecture, which allowed us to understand the agents, system entities, processes, and standards involved in an accountable network. That accountability analysis, along with a multi-app case study, provided us with our basis for understanding the security challenges of the SDN architecture that we investigated throughout the remainder of the dissertation.

Next, we identified the cross-app poisoning (CAP) problem, in which confused deputy attacks by malicious apps can influence benign apps to take actions on their behalf. We showed how the control plane lacked information flow control (IFC) mechanisms. We identified where CAP vulnerabilities occurred in spite of least-privilege RBAC enforcement. We defended against CAP vulnerabilities

through data provenance tracking for IFC, and we designed PROVSDN to enforce IFC and to serve as a reference monitor.

In addition to cross-app-based vulnerabilities, we also investigated a class of cross-plane vulnerabilities that leverage missing event handling. We showed the various ways in which data plane hosts can influence control plane decisions with respect to how events are used and executed. We designed an approach to analyzing apps' event use that identifies likely missing event handling. We created the event flow graph data structure to aid in the identification of event-based vulnerabilities and to succinctly represent control plane activities. We designed EVENTSCOPE to generate event flow graphs and to find vulnerabilities, and we discovered 14 new vulnerabilities in the ONOS controller.

Finally, we revisited the use of data provenance in the control plane for the purpose of causal analysis. We mitigated the dependency explosion problem through fine-grained data partitioning and execution partitioning techniques for control plane objects and event listeners, respectively. We mitigated the incomplete dependency problem through the inclusion of a data plane model. We designed PICO SDN to aid in the analysis of SDN attacks.

7.2 Overall Takeaways

We now revisit the research questions posed in Chapter 1. We consider how our dissertation's contributions provide solutions to those questions and the overall takeaways of our impacts and insights.

Network application composability We found that the SDN architecture in use today, as implemented in network operating systems such as ONOS, does not provide a clear mechanism for inter-process communication (IPC) among apps.¹ As a result, apps must communicate through northbound control plane API interfaces or through event notifications. In contrast to mobile or host operating systems with well-defined IPC interfaces [7], network operating systems often rely on a shared state design in which apps communicate through the writing and reading of shared data structures. We discovered that this can induce strong data dependencies across apps from an information flow perspective. From a security perspective, it allows untrusted lower-integrity apps to poison the state of trusted higher-integrity apps. Our takeaway is that *the use and generation of information must be tracked across apps to prevent “confused deputy” attacks.*

The tight coupling among apps through a shared state design is not surprising, given that the use of network abstractions is prevalent in SDN network operating systems. For instance, a controller core service can use information derived from Packet objects to generate Host control plane objects; apps can use higher abstractions (*i.e.*, Host) without duplicating functionality (*i.e.*, the Packet→Host

¹Early SDN network operating systems, such as Rosemary [14], provide IPC between an app and the controller's core services only. That increases apps' isolation but does not provide cross-app communication interfaces.

transformation). A shared state design could be problematic for security enforcement in mobile or host operating system contexts in which apps can operate relatively independently; to address that concern, authorization and permission systems such as role-based access control (RBAC) can enforce app isolation when apps work independently. For network operating systems, we found that a shared state design works well for SDN because the network operating system is ultimately responsible for the behavior of one shared resource: the network’s data plane. However, this shared state design suggests that RBAC is not the right access control model when data dependencies are high. Nevertheless, we found that authorization and permission systems for control plane objects in prior work [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21] all used some variation of RBAC. Our takeaway is that *RBAC is problematic for security enforcement in the SDN architecture’s highly dependent shared state design because RBAC does not track how data are used after permission has been granted.*

The design of the SDN architecture assumes that apps work together in a collaborative and composable way, but we found that such apps often have competing functionalities and work against each other in practice. Although prior work [17, 114] has studied malicious apps in isolation, we found that the combination of apps creates complex and nontrivial vulnerabilities. Such vulnerabilities are caused by missing event handling among apps, which resulted in the absence of expected control flow execution and the presence of unexpected control flow execution. Our takeaway is that *the security analysis of apps must occur at a system-wide level in order to reveal the security ramifications of composable apps.* This takeaway suggests that app developers should “design defensively” by considering all possible uses of the events to which apps subscribe.

Control and data plane decoupling Although the SDN architecture decouples the control and data planes [1], we found that hosts on the data plane can have an outsized effect on the control plane’s behavior. This impact results from reactive control planes’ use of the data plane to make informed decisions about the network’s current state. That challenge is generally hard to defend against because the underlying host-generated network protocol messages, such as ARP, do not offer authentication by default. We found that malicious hosts can take advantage of data plane information that is used as input into control plane decision-making, and that, in turn, changes the data plane configuration (*i.e.*, it creates a data plane→control plane→data plane pattern of information and control flow). Our takeaway is that *the control and data planes are, in practice, tightly integrated, which can cause trust vulnerabilities in reactive control planes.*

We also discovered that the control plane can have indirect causal effects on itself through the data plane (*i.e.*, a control plane→data plane→control plane pattern of information and control flow). For instance, reactive forwarding will emit packets out of a forwarding device, and those packets will trigger subsequent control plane actions when received by another forwarding device. That indirect causal influence via the data plane impacted our decision to include a data plane model in our

control plane provenance. Our takeaway is that *a control plane activity can use the data plane to cause another control plane activity to occur, which must be tracked for information and control flow purposes.*

Control plane insight We found that static analysis techniques offer insight into many possible control and data dependencies, and thereby aid in the systematic identification of vulnerabilities, but cannot provide a history of how specific vulnerabilities were actually exploited in attacks. We also found that provenance techniques offer insight into the specific control and data dependencies used during an execution, and thereby reveal root causes of attacks, but may not be able to diagnose corner cases of rarely-used control and data flow paths in execution. However, the data provenance techniques used in PROVSDN and PICO SDN and the static analysis techniques used in EVENTSCOPE complement each other in improving our understanding of the control plane during the runtime, pre-runtime, and post-runtime stages. Our takeaway is that *the combination of static analysis and data provenance approaches allows us to better understand and analyze the control plane, and that improved understanding of the control plane enables us to improve the security posture of the SDN architecture.*

We found that although the reference monitor concept has already been proposed for SDN [23], it had not previously been implemented in any popular network operating systems, such as ONOS [25]. A reference monitor can centrally interpose across relevant requests and can ensure completeness of mediation for access control or insight. In ONOS, API requests are not centrally managed, so instrumentation of all API calls is required to ensure completeness. In PROVSDN, we demonstrated the need to provide a reference monitor over API requests. In PICO SDN, we demonstrated the additional need to provide a reference monitor over event dispatching and event listening. Our takeaway is that *a reference monitor for the control plane must reside at the API and event boundaries in order to ensure oversight completeness.*

We found that we need semantically aware and precise abstractions for a network in order to reason effectively about what the network has done. The general data structures and abstractions found in static analysis, such as call graphs and control flow graphs, are necessary but not sufficient for reasoning about event-based vulnerabilities. Similarly, we found that our data provenance model for the SDN control and data planes required that we specify data and processes at a fine granularity to avoid dependency explosion issues (*i.e.*, false positives) and that we had to include the data plane in our causal model to avoid incomplete dependency issues (*i.e.*, false negatives). We also found that the modeling of principal agents in SDN can be challenging. Such agency attribution is necessary in a shared state design, but we found that assigning agency to hosts was a poor modeling option because such hosts' network identifiers are easily spoofed; instead, we relied upon switch ports as principal agents. Our takeaway is that *our abilities to reason about the control plane are significantly*

influenced by the modeling, abstraction, and agency choices used.

7.3 Future Research

The work in this dissertation can be extended in several directions.

Malware-based attacks on SDN Recent malware campaigns, such as NotPetya [271], leveraged vulnerable hosts on open networks in order to spread laterally to infect other hosts. SDN as a network security service has been shown to provide promising mechanisms for preventing such lateral movement through the enforcement of fine-grained, least-privilege data plane access control [4]. However, in spite of such enforcement, we anticipate that the programmable control plane will become a target for malware developers who leverage cross-plane vulnerabilities.

Although we have not seen publicly disclosed examples of exploitation of SDN controller vulnerabilities in practice, the building blocks for such exploitation by malware already exist. First, prior work has demonstrated that malicious hosts can tell whether or not they are residing on an SDN-controlled network [66, 67, 72], and that if the network is SDN-controlled, the host can determine which controller [64, 65] and apps [47] are being used. Second, this dissertation showed how the policies of data plane access control apps, such as the acl app in the ONOS controller, can be bypassed with several packets. We hypothesize that SDN-aware malware would be able to leverage such fingerprinting and packet-generating capabilities to defeat lateral movement prevention mechanisms from the narrow (but powerful) data plane attack vector.

We envision that future research in this area will require a better understanding of how hosts interact with and influence the control plane, particularly for detection and prevention purposes. We imagine that cross-layer provenance capabilities, such as systems that integrate network-level, host-level, and application-level provenance, will be able to provide practitioners with better insight and analysis than individual layers alone can provide.

SDN attack patterns We foresee that the abstractions and models that we designed for the event flow graph and control plane provenance will have broader applicability in the understanding and analysis of network attacks.

Although we used the event flow graph for the detection of missing event handling, the same structure may be amenable to use in the identification of other attack classes. For instance, the presence of several paths in an event flow graph may be indicative of race conditions for causal chains of event dispatches and event listens by apps. As a future research direction, we anticipate that other attack classes may be mapped into the event flow graph structure and thus would allow us to detect and fix those vulnerabilities prior to runtime.

Control plane provenance graphs provide a dataset from which one can find common patterns of repetitive and commonly used control flows in the control plane. Deviations from such patterns may be an indication of an anomalous behavior that practitioners can use to understand whether such behavior is an attack that is causing undesirable control plane behavior. Limitations exist in the use of pattern recognition techniques for the identification of outliers [272], and we anticipate that the semantics of SDN and the control plane may be useful in mitigating such limitations.

Control plane policy generation Verification-based approaches for SDN security can check against well-known invariants of network properties or against desired network policy, but the generation of such properties or policies is challenging and may not reflect the intended policy. We imagine that pattern recognition over control plane provenance graphs may be able to inform the control plane policy generation process. Such pattern recognition could inform practitioners in the policy-writing process and result in policies that reflect how the network is actually used in practice. This research direction would complement existing work on network verification.

Programmable data planes The extensibility of the SDN architecture has recently created additional abstractions. OpenFlow [27] operates as a configuration protocol for the data plane but is limited to a static set of match fields based on well-known network protocols. Programmable data planes allow for arbitrary fields to be matched and forwarding actions to be taken. Languages for programmable data planes, such as P4 [36], provide capabilities beyond the control plane to extend the network's programmability. Just as we implemented control plane insight and vulnerability detection for this dissertation, we imagine that similar data plane insight and vulnerability detection mechanisms will be required for attack prevention and analysis. Initial work in the networking community has started to explore the data plane execution [273] and buggy data plane programs [274].

APPENDIX A

PUBLICATIONS RELATED TO THE DISSERTATION

The following peer-reviewed publications are related to this dissertation. Authors are affiliated with the University of Illinois at Urbana-Champaign unless otherwise noted.

- Chapter 3 is based on “Towards an Accountable Software-Defined Networking Architecture,” which was accepted to the 2017 IEEE Conference on Network Softwarization (NetSoft ’17) and was co-authored by Andrew Miller, Adam Bates, and William H. Sanders. The Version of Record can be found at <https://doi.org/10.1109/NETSOFT.2017.8004206>.
- Chapter 4 is based on “Cross-App Poisoning in Software-Defined Networking,” which was accepted to the 2018 ACM Conference on Computer and Communications Security (CCS ’18) and was co-authored by Samuel Jero (MIT Lincoln Laboratory), Anne Edmundson (Princeton University), Qi Wang, Richard Skowyra (MIT Lincoln Laboratory), James Landry (MIT Lincoln Laboratory), Adam Bates, William H. Sanders, Cristina Nita-Rotaru (Northeastern University), and Hamed Okhravi (MIT Lincoln Laboratory). The Version of Record can be found at <https://doi.org/10.1145/3243734.3243759>.
- Chapter 5 is based on “Automated Discovery of Cross-Plane Event-Based Vulnerabilities in Software-defined Networking,” which was accepted to the 2020 Internet Society Network and Distributed System Security Symposium (NDSS ’20) and was co-authored by Samuel Jero (MIT Lincoln Laboratory), Richard Skowyra (MIT Lincoln Laboratory), Steven R. Gomez (MIT Lincoln Laboratory), Adam Bates, William H. Sanders, and Hamed Okhravi (MIT Lincoln Laboratory). The Version of Record can be found at <https://doi.org/10.14722/ndss.2020.24080>.
- Chapter 6 is based on “Causal Analysis for Software-Defined Networking Attacks,” which is in submission at the time of this writing and was co-authored by Samuel Jero (MIT Lincoln Laboratory), Richard Skowyra (MIT Lincoln Laboratory), Adam Bates, William H. Sanders, and Hamed Okhravi (MIT Lincoln Laboratory).

In reference to IEEE copyrighted material that is used with permission in this dissertation, the IEEE does not endorse any of the University of Illinois at Urbana-Champaign’s products or ser-

vices. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE-copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

APPENDIX B

PROVSDN

B.1 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 [19]. We considered the API level permissions in our RBAC analysis in Section 4.5.1, 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 [166] that implemented such permissions at the time of writing.

B.1.1 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 [275]. 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 [19]. Such a manifest subsequently is included with the app and is verified when the app is installed [276].

In addition to our static analysis script (see Section 4.5.1.4) 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.

Table B.1: Partial RBAC Model for Security-Mode ONOS and Included ONOS Apps.

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

B.1.2 App, Permission, and Object Details

Table B.1 enumerates the specific apps, read permissions, write permissions, and objects that we used in our CAP model for Security-Mode ONOS.

B.2 Selected Code for Reactive Forwarding App

Figure B.1 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 B.1) and to process such packets (Line 9, Figure B.1), in addition to the FLOWRULE_WRITE permission to emit flow rules into the data plane (Line 17, Figure B.1). We also permitted fwd to have the APP_* (Line 4, Figure B.1), 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 B.1), 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 4.5.3, trigger poisons such data before they arrive to fwd (Line 9, Figure B.1).

B.3 W3C PROV-DM Representations

Table B.2 summarizes the visual representations of the W3C PROV data model's provenance objects and relations [184]. 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.

B.4 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 }

```




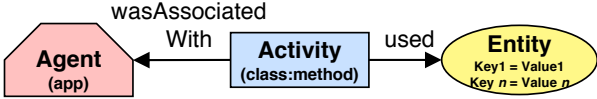
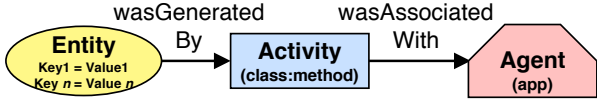
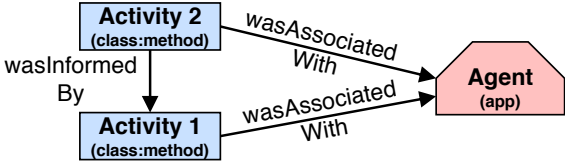
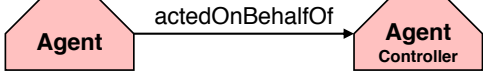
Figure B.1: Selected reactive forwarding app code during app activation and during packet processing for inserting flow rules. Lines with permissioned calls are highlighted in gray.

B.4.1 Java-Based Open-Source Controllers

In addition to ONOS, Floodlight [28], SE-Floodlight [15], and OpenDaylight [30] 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.

Table B.2: SDN Shared Control Plane State Semantics Using W₃C PROV-DM.

Object or Event	W ₃ C 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	

B.4.2 Python-Based Open-Source Controllers

Several SDN controllers, including Ryu [29] and POX [82], 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 is difficult to support internal 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.

B.4.3 C/C++-Based Open-Source Controllers

Controllers written in C or C++, such as Rosemary [14] and NOX [31], 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.

B.4.4 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 [277]. However, possible future work could leverage verbose log files to gain insight into interactions between the controller and apps.

APPENDIX C

EVENTSCOPE

C.1 ONOS Application Structure

C.1.1 App Components

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

- **Internal variables (lines 3–8):** Internal variables maintain the app’s state, which includes references to data store objects and core controller services. In `sampleApp`, references to the host, packet, and flow rule services are created, along with the instantiations of the host (event) listener and the packet processor.
- **Activation and deactivation methods (lines 9–19):** The activation method is called once when the app is activated; similarly, the deactivation method is called once during deactivation. During activation and deactivation, the app registers and deregisters components that it needs, such as event listeners and packet processors. In `sampleApp`, the host event listener and packet processor are registered and deregistered.
- **Event listeners (lines 20–29):** Event listeners listen for an event kind of interest and take further action, often based on the event type. Event listeners may call other methods within the app to perform a desired functionality. In `sampleApp`, the host event listener executes `event()` when it receives a `HostEvent` (line 22). It handles the `HOST_ADDED` type by calling the internal method `internalMethod1()` (line 25). Note that all other `HostEvent` event types (e.g., `HOST_REMOVED`) are not handled.
- **Packet processors (lines 30–37):** Packet processors function much like to event listeners by listening for incoming data plane packets and taking appropriate actions. In `sampleApp`, the

```

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

```

Figure C.1: Abbreviated code structure of an example ONOS network application, sampleApp, written in Java.

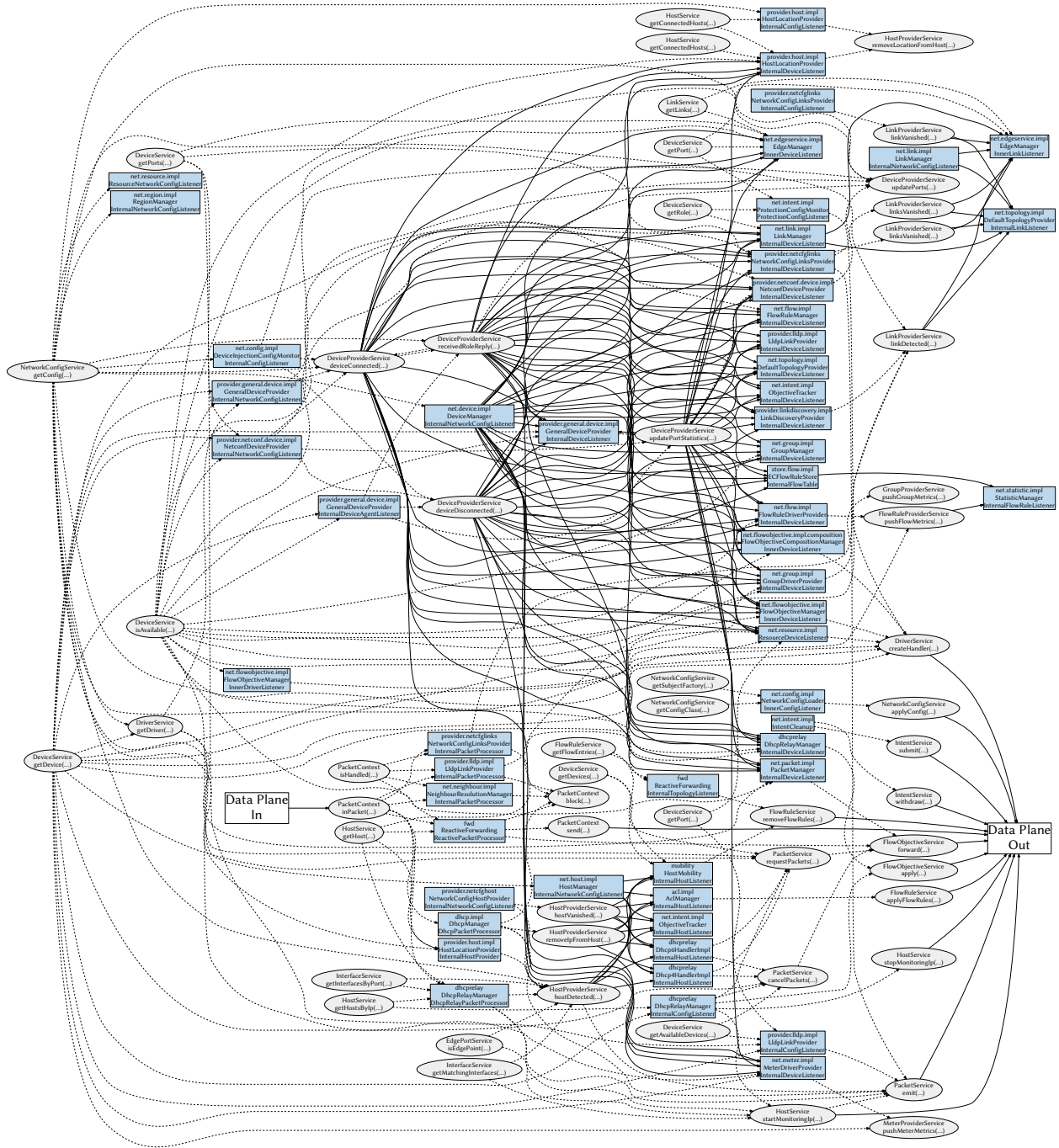


Figure C.2: Event flow graph of ONOS with core service components and several apps (*i.e.*, acl, fwd, mobility, dhcp, and dhcprelay). Blue rectangles represent event listeners and packet processors, gray ellipses represent API methods, bold edges represent event dispatches, and dashed edges represent API calls. (For simplicity, event types are reduced to a single edge of the event type's respective event kind.)

packet processor executes `process()` when it receives a packet (line 32) and subsequently after execution calls the internal method `internalMethod2()` (line 34).

- **App internal methods (lines 38–45):** App internal methods handle the main functionality of the app. They may read from core services (*i.e.*, API read calls), write to core services (*i.e.*, API write calls), or dispatch new events. In `sampleApp`, `internalMethod1()` calls `internalMethod2()`. New flow rules are generated as a result of the calling of `internalMethod2()` (line 44).

C.1.2 App Analysis

We explain how `sampleApp` would be analyzed within `EVENTSCOPE`.

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

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

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

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

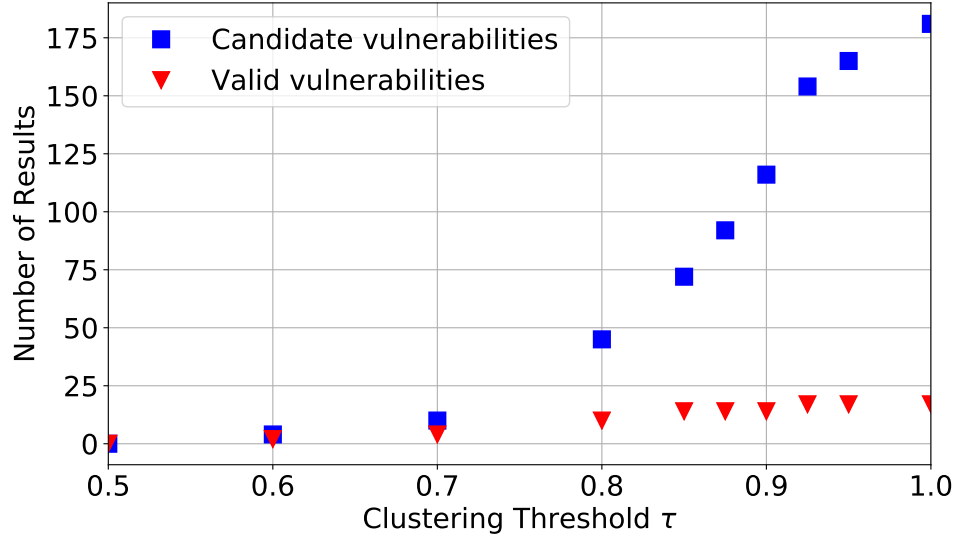


Figure C.3: ONOS apps' candidate and valid vulnerabilities as a function of clustering threshold τ (using SimRank [213]).

C.2 ONOS Event Flow Graph Example

Figure C.2 shows the ONOS event flow graph with the controller's core services, the access control app (acl), the reactive forwarding app (fwd), the host mobility app (mobility), and the DHCP apps (dhcp and dhcprelay).

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

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

C.3 Number of Clusters and Detection Rate

Figure C.3 shows the effect of choosing different values for the event use clustering threshold τ (*i.e.*, changing the number of clusters) on the detection rate for the number of candidate vulnera-

bilities (Section 5.4.1) and valid vulnerabilities (Section 5.5.2) for ONOS v1.14.0 [26]. We note an inflection point of candidate vulnerabilities near $\tau = 0.90$, which is the threshold that we used throughout our evaluation.

REFERENCES

- [1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.
(Cited on pages 1, 2, 7, 27, 36, 65, and 131.)
- [2] M. Cooney, “What is SDN and where software-defined networking is going,” Apr. 2019, *Network World*. [Online]. Available: <https://www.networkworld.com/article/3209131/what-sdn-is-and-where-its-going.html>
(Cited on page 1.)
- [3] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, “Let SDN be your eyes: Secure forensics in data center networks,” in *Proceedings of the 2014 Workshop on Security of Emerging Networking Technologies (SENT ’14)*. Internet Society, Apr. 2014.
(Cited on pages 1, 27, 32, 64, and 128.)
- [4] S. R. Gomez, S. Jero, R. Skowyra, J. Martin, P. Sullivan, D. Bigelow, Z. Ellenbogen, B. C. Ward, H. Okhravi, and J. W. Landry, “Controller-oblivious dynamic access control in software-defined networks,” in *Proceedings of the 2019 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’19)*. IEEE, June 2019, pp. 447–459.
(Cited on pages 1, 102, and 133.)
- [5] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, “FRESCO: Modular composable security services for software-defined networks,” in *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS ’13)*. Internet Society, Feb. 2013.
(Cited on pages 1, 19, and 63.)
- [6] D. Kreutz, F. M. Ramos, and P. Veríssimo, “Towards secure and dependable software-defined networks,” in *Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’13)*. ACM, 2013, pp. 55–60.
(Cited on pages 1, 13, 29, 38, 41, and 42.)
- [7] W. Enck, M. Ongtang, and P. McDaniel, “Understanding Android security,” *IEEE Security & Privacy*, vol. 7, no. 1, pp. 50–57, Jan. 2009.
(Cited on pages 2 and 130.)
- [8] R. Sherwood, G. Gibb, K.-K. Yap, M. Casado, N. McKeown, and G. Parulkar, “FlowVisor: A network virtualization layer,” OpenFlow, Tech. Rep., 2009.
(Cited on page 2.)

- [9] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, “Survey on network virtualization hypervisors for software defined networking,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 655–685, 2016.
(Cited on page 2.)
- [10] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang, “Towards a secure controller platform for OpenFlow applications,” in *Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’13)*. ACM, 2013, pp. 171–172.
(Cited on pages 2, 20, 21, and 131.)
- [11] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, “A security enforcement kernel for OpenFlow networks,” in *Proceedings of the 2012 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’12)*. ACM, 2012, pp. 121–126.
(Cited on pages 2, 9, 14, 17, 20, 21, 22, and 131.)
- [12] S. Scott-Hayward, C. Kane, and S. Sezer, “OperationCheckpoint: SDN application control,” in *Proceedings of the 2014 IEEE International Conference on Network Protocols (ICNP ’14)*. IEEE, Oct 2014, pp. 618–623.
(Cited on pages 2, 20, 21, and 131.)
- [13] S. Matsumoto, S. Hitz, and A. Perrig, “Fleet: Defending SDNs from malicious administrators,” in *Proceedings of the 2014 ACM Workshop on Hot Topics in Networks (HotNets ’14)*. ACM, 2014, pp. 103–108.
(Cited on pages 2, 14, 19, 21, 22, 30, and 131.)
- [14] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, “Rosemary: A robust, secure, and high-performance network operating system,” in *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS ’14)*. ACM, 2014, pp. 78–89.
(Cited on pages 2, 9, 20, 21, 23, 38, 41, 63, 130, 131, and 142.)
- [15] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, “Securing the software-defined network control layer,” in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS ’15)*. Internet Society, Feb. 2015.
(Cited on pages 2, 9, 20, 21, 22, 29, 30, 31, 32, 35, 36, 38, 41, 60, 63, 112, 131, and 140.)
- [16] H. Padekar, Y. Park, H. Hu, and S.-Y. Chang, “Enabling dynamic access control for controller applications in software-defined networks,” in *Proceedings of the 2016 ACM Symposium on Access Control Models and Technologies (SACMAT ’16)*. ACM, 2016, pp. 51–61.
(Cited on pages 2, 20, 21, 38, and 131.)
- [17] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, and X. Chen, “SDNShield: Reconciliating configurable application permissions for SDN app markets,” in *Proceedings of the 2016 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’16)*. IEEE, 2016, pp. 121–132.
(Cited on pages 2, 14, 17, 19, 20, 21, 23, 43, 61, 63, and 131.)

- [18] Y. Tseng, M. Pattaranantakul, R. He, Z. Zhang, and F. Naït-Abdesselam, “Controller DAC: Securing SDN controller with dynamic access control,” in *Proceedings of the 2017 IEEE International Conference on Communications (ICC ’17)*. IEEE, May 2017, pp. 1–6.
(Cited on pages 2, 20, 21, 38, and 131.)
- [19] C. Yoon, S. Shin, P. Porras, V. Yegneswaran, H. Kang, M. Fong, B. O’Connor, and T. Vachuska, “A security-mode for carrier-grade SDN controllers,” in *Proceedings of the 2017 Annual Computer Security Applications Conference (ACSAC ’17)*, 2017, pp. 461–473.
(Cited on pages 2, 9, 20, 21, 38, 39, 46, 63, 131, and 137.)
- [20] B. Toshniwal, K. D. Joshi, P. Shrivastava, and K. Kataoka, “BEAM: Behavior-based access control mechanism for SDN applications,” in *Proceedings of the 2019 IEEE International Conference on Computer Communication and Networks (ICCCN ’19)*. IEEE, 2019, pp. 1–2.
(Cited on pages 2, 20, 21, and 131.)
- [21] F. Klaedtke, G. O. Karame, R. Bifulco, and H. Cui, “Access control for SDN controllers,” in *Proceedings of the 2014 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’14)*. ACM, 2014, pp. 219–220.
(Cited on pages 2 and 131.)
- [22] S. Hong, L. Xu, H. Wang, and G. Gu, “Poisoning network visibility in software-defined networks: New attacks and countermeasures,” in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS ’15)*. Internet Society, Feb. 2015.
(Cited on pages 3, 14, 18, 21, 22, 23, 66, 70, 95, 96, 99, 101, 112, 119, 121, 125, 126, and 127.)
- [23] D. Gkounis, F. Klaedtke, R. Bifulco, and G. O. Karame, “Cases for including a reference monitor to SDN,” in *Proceedings of the 2016 ACM Special Interest Group on Data Communication Conference (SIGCOMM ’16)*. ACM, 2016, pp. 599–600.
(Cited on pages 3 and 132.)
- [24] R. Kissel, “Glossary of key information security terms,” National Institute of Standards and Technology, Tech. Rep. NISTIR 7298, May 2013. [Online]. Available: <http://dx.doi.org/10.6028/NIST.IR.7298r2>
(Cited on pages 3, 15, 19, and 29.)
- [25] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, and G. Parulkar, “ONOS: Towards an open, distributed SDN OS,” in *Proceedings of the 2014 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’14)*. ACM, 2014.
(Cited on pages 4, 8, 9, 10, 22, 28, 30, 31, 38, 39, 46, 65, 68, 101, 105, 108, and 132.)
- [26] Open Networking Foundation, “Github – opennetworkinglab/onos at 1.14.0,” 2019. [Online]. Available: <https://github.com/opennetworkinglab/onos/tree/onos-1.14>
(Cited on pages 4, 76, 120, and 148.)
- [27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

(Cited on pages 7, 10, 13, and 134.)

- [28] Floodlight, 2018. [Online]. Available: <http://www.projectfloodlight.org/>
(Cited on pages 8, 9, 16, 20, 31, 60, 65, 69, 95, 105, 108, 120, and 140.)
- [29] Ryu SDN Framework Community, “Ryu SDN framework,” Aug. 2018. [Online]. Available: <http://osrg.github.io/ryu/>
(Cited on pages 8, 9, 16, 61, and 141.)
- [30] OpenDaylight, “OpenDaylight,” Aug. 2018. [Online]. Available: <https://www.opendaylight.org/>
(Cited on pages 8, 9, 16, 22, 38, 60, 65, 105, 108, 120, and 140.)
- [31] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, “NOX: Towards an operating system for networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.
(Cited on pages 9, 20, and 142.)
- [32] Open Networking Foundation, “System components – ONOS,” 2019. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/System+Components>
(Cited on pages 9 and 81.)
- [33] Open Networking Foundation, “Distributed primitives – ONOS,” 2019. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/Distributed+Primitives>
(Cited on page 9.)
- [34] R. Enns, “NETCONF configuration protocol,” Internet Engineering Task Force (IETF), RFC 4741, Dec. 2006. [Online]. Available: <https://tools.ietf.org/html/rfc4741>
(Cited on page 10.)
- [35] E. Haleplidis, J. Hadi Salim, J. M. Halpern, S. Hares, K. Pentikousis, K. Ogawa, W. Wang, S. Denazis, and O. Koufopavlou, “Network programmability with ForCES,” *IEEE Communications Surveys Tutorials*, vol. 17, no. 3, pp. 1423–1440, 2015.
(Cited on page 10.)
- [36] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, July 2014.
(Cited on pages 10, 13, 115, and 134.)
- [37] Hewlett-Packard Enterprise, “HPE SDN app store,” <https://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps>, Aug. 2018.
(Cited on pages 12, 37, and 40.)
- [38] Open Networking Foundation, “GitHub – onos/apps at 1.10.0,” 2018. [Online]. Available: <https://github.com/opennetworkinglab/onos/tree/1.10.0/apps>
(Cited on pages 12 and 46.)

- [39] Open Networking Foundation, “GitHub – onos-app-samples,” 2018. [Online]. Available: <https://github.com/opennetworkinglab/onos-app-samples>
(Cited on page 12.)
- [40] Linux Foundation, “Open vSwitch,” 2018. [Online]. Available: <https://www.openvswitch.org/>
(Cited on pages 13, 18, and 59.)
- [41] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 2010 ACM Workshop on Hot Topics in Networks (HotNets ’10)*. ACM, 2010.
(Cited on pages 13, 86, and 121.)
- [42] Open Networking Foundation, “OpenFlow v1.3.0,” June 2012. [Online]. Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf>
(Cited on pages 13, 24, 30, and 31.)
- [43] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu, “Unexpected data dependency creation and chaining: A new attack to SDN,” in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P ’20)*. IEEE, May 2020.
(Cited on pages 14, 19, 25, 99, and 126.)
- [44] B. E. Ujcich, S. Jero, R. Skowyra, S. R. Gomez, A. Bates, W. H. Sanders, and H. Okhravi, “Automated discovery of cross-plane event-based vulnerabilities in software-defined networking,” in *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS ’20)*. Internet Society, Feb. 2020.
(Cited on pages 14, 19, 25, 99, 101, 105, 110, 119, 121, 126, and 127.)
- [45] J. Cao, R. Xie, K. Sun, Q. Li, G. Gu, and M. Xu, “When match fields do not need to match: Buffered packets hijacking in SDN,” in *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS ’20)*. Internet Society, Feb. 2020.
(Cited on pages 14, 18, and 99.)
- [46] J. Xu, L. Wang, C. Song, and Z. Xu, “An effective table-overflow attack and defense in software-defined networking,” in *Proceedings of the 2019 IEEE Local Computer Networks Conference (LCN ’19)*. IEEE, 2019, pp. 10–17.
(Cited on pages 14, 15, 21, and 24.)
- [47] J. Cao, Z. Yang, K. Sun, Q. Li, M. Xu, and P. Han, “Fingerprinting SDN applications via encrypted control traffic,” in *Proceedings of the 2019 International Symposium on Research in Attacks, Intrusions and Defenses (RAID ’19)*. USENIX Association, Sep. 2019, pp. 501–515.
(Cited on pages 14, 17, and 133.)
- [48] E. Marin, N. Buccioli, and M. Conti, “An in-depth look into SDN topology discovery mechanisms: Novel attacks and practical countermeasures,” in *Proceedings of the 2019 ACM Conference on Computer and Communications Security (CCS ’19)*. ACM, 2019.
(Cited on pages 14, 18, 99, 101, and 112.)

- [49] R. Hanmer, S. Liu, L. Jagadeesan, and M. R. Rahman, “Death by babble: Security and fault tolerance of distributed consensus in high-availability softwarized networks,” in *Proceedings of the 2019 IEEE International Conference on Network Softwarization (NetSoft ’19)*. IEEE, June 2019, pp. 266–270.
(Cited on pages 14, 16, and 93.)
- [50] D. Smyth, D. O’Shea, V. Cionca, and S. McSweeney, “Attacking distributed software-defined networks by leveraging network state consistency,” *Computer Networks*, vol. 156, pp. 9 – 19, 2019.
(Cited on pages 14 and 16.)
- [51] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, “Cross-app poisoning in software-defined networking,” in *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS ’18)*. ACM, 2018.
(Cited on pages 14, 17, 20, 21, 22, 67, 70, 71, 82, 96, 97, 99, 100, 104, 106, 112, 121, 122, 125, and 127.)
- [52] K. Thimmaraju, L. Schiff, and S. Schmid, “Outsmarting network security with SDN teleportation,” in *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroSecP ’17)*. IEEE, 2017, pp. 563–578.
(Cited on pages 14 and 17.)
- [53] J. Cao, Q. Li, R. Xie, K. Sun, G. Gu, M. Xu, and Y. Yang, “The CrossPath attack: Disrupting the SDN control channel via shared links,” in *Proceedings of the 2019 USENIX Security Symposium (Security ’19)*. USENIX Association, Aug. 2019.
(Cited on pages 14 and 17.)
- [54] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai, “Control plane reflection attacks in SDNs: New attacks and countermeasures,” in *Proceedings of the 2018 International Symposium on Research in Attacks, Intrusions and Defenses (RAID ’18)*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., 2018, pp. 161–183.
(Cited on pages 14, 15, 21, and 24.)
- [55] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, “Effective topology tampering attacks and defenses in software-defined networks,” in *Proceedings of the 2018 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’18)*. IEEE, June 2018, pp. 374–385.
(Cited on pages 14, 18, 66, 70, 95, 96, 99, 101, 112, 119, 126, and 127.)
- [56] K. Thimmaraju, B. Shastri, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, “Taking control of SDN-based cloud systems via the data plane,” in *Proceedings of the 2018 ACM SIGCOMM Symposium on SDN Research (SOSR ’18)*. ACM, 2018.
(Cited on pages 14 and 18.)
- [57] S. Deng, X. Gao, Z. Lu, and X. Gao, “Packet injection attack and its defense in software-defined networks,” *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 695–705, 2018.
(Cited on pages 14, 19, 21, and 23.)

- [58] S. Jero, W. Koch, R. Skowrya, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, “Identifier binding attacks and defenses in software-defined networks,” in *Proceedings of the 2017 USENIX Security Symposium (Security ’17)*. USENIX Association, 2017.
(Cited on pages 14, 18, 21, 22, 70, 95, 96, 101, and 110.)
- [59] M. Ambrosin, M. Conti, F. De Gaspari, and R. Poovendran, “LineSwitch: Tackling control plane saturation attacks in software-defined networking,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1206–1219, 2017.
(Cited on pages 14, 15, 21, and 23.)
- [60] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, “AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks,” in *Proceedings of the 2013 ACM Conference on Computer and Communications Security (CCS ’13)*. ACM, 2013, pp. 413–424.
(Cited on pages 14, 15, 21, and 23.)
- [61] S. Liu, M. K. Reiter, and V. Sekar, “Flow reconnaissance via timing attacks on SDN switches,” in *Proceedings of the 2017 IEEE International Conference on Distributed Computing Systems (ICDCS ’17)*. IEEE, 2017, pp. 196–206.
(Cited on pages 14 and 17.)
- [62] P. Zhang, “Towards rule enforcement verification for software defined networks,” in *Proceedings of the 2017 IEEE International Conference on Computer Communications (INFOCOM ’17)*. IEEE, 2017, pp. 1–9.
(Cited on pages 14, 18, 21, and 23.)
- [63] D. Smyth, V. Cionca, S. McSweeney, and D. O’Shea, “Exploiting pitfalls in software-defined networking implementation,” in *Proceedings of the 2016 IEEE International Conference on Cyber Security and Protection of Digital Services (CyberSecurity ’16)*. IEEE, 2016, pp. 1–8.
(Cited on pages 14 and 15.)
- [64] M. Zhang, J. Hou, Z. Zhang, W. Shi, B. Qin, and B. Liang, “Fine-grained fingerprinting threats to software-defined networks,” in *Proceedings of the 2017 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom ’17)*. IEEE, Aug. 2017, pp. 128–135.
(Cited on pages 14, 17, 70, and 133.)
- [65] A. Azzouni, O. Braham, T. M. T. Nguyen, G. Pujolle, and R. Boutaba, “Fingerprinting Open-Flow controllers: The first step to attack an SDN control plane,” in *Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM ’16)*. IEEE, Dec. 2016, pp. 1–6.
(Cited on pages 14, 16, 70, and 133.)
- [66] R. Bifulco, H. Cui, G. O. Karame, and F. Klaedtke, “Fingerprinting software-defined networks,” in *Proceedings of the 2015 IEEE International Conference on Network Protocols (ICNP ’15)*. IEEE, 2015, pp. 453–459.
(Cited on pages 14, 16, and 133.)

- [67] J. Sonchack, A. Dubey, A. J. Aviv, J. M. Smith, and E. Keller, “Timing-based reconnaissance and defense in software-defined networks,” in *Proceedings of the 2016 Annual Computer Security Applications Conference (ACSAC ’16)*, 2016, pp. 89–100.
(Cited on pages 14, 16, 21, 24, and 133.)
- [68] T. Alharbi, M. Portmann, and F. Pakzad, “The (in)security of topology discovery in software defined networks,” in *Proceedings of the 2015 IEEE Local Computer Networks Conference (LCN ’15)*. IEEE, 2015, pp. 502–505.
(Cited on pages 14, 18, 21, and 22.)
- [69] C. Röpke and T. Holz, “SDN rootkits: Subverting network operating systems of software-defined networks,” in *Proceedings of the 2015 International Symposium on Research in Attacks, Intrusions and Defenses (RAID ’15)*. Springer-Verlag New York, Inc., 2015, pp. 339–356.
(Cited on pages 14, 17, and 27.)
- [70] Y. Zhou, K. Chen, J. Zhang, J. Leng, and Y. Tang, “Exploiting the vulnerability of flow table overflow in software-defined network: Attack model, evaluation, and defense,” in *Security and Communication Networks*, Z. Cai, Ed., vol. 2018. Hindawi, 2018.
(Cited on pages 14 and 17.)
- [71] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, “How to detect a compromised SDN switch,” in *Proceedings of the 2015 IEEE International Conference on Network Softwarization (NetSoft ’15)*. IEEE, April 2015, pp. 1–6.
(Cited on pages 14, 18, 19, 21, and 22.)
- [72] S. Shin and G. Gu, “Attacking software-defined networks: A first feasibility study,” in *Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’13)*. ACM, 2013, pp. 165–166.
(Cited on pages 14, 16, and 133.)
- [73] J. Hizver, “Taxonomic modeling of security threats in software defined networking,” in *Proceedings of the 2015 BlackHat Conference*, 2015.
(Cited on page 13.)
- [74] K. Benton, L. J. Camp, and C. Small, “OpenFlow vulnerability assessment,” in *Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’13)*. ACM, 2013, pp. 151–152.
(Cited on page 13.)
- [75] R. Klöti, V. Kotronis, and P. Smith, “OpenFlow: A security analysis,” in *Proceedings of the 2013 IEEE International Conference on Network Protocols (ICNP ’13)*. IEEE, 2013, pp. 1–6.
(Cited on page 13.)
- [76] S. Scott-Hayward, S. Natarajan, and S. Sezer, “A survey of security in software defined networks,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 623–654, 2016.
(Cited on pages 13 and 38.)

- [77] A. B. Martin, L. Marinos, E. Rekleitis, G. Spanoudakis, and N. Petroulakis, “Threat landscape and good practice guide for software defined networks/5G,” European Union Agency for Network and Information Security, Tech. Rep., Dec. 2015.
(Cited on page 13.)
- [78] S. Khan, A. Gani, A. W. Abdul Wahab, M. Guizani, and M. K. Khan, “Topology discovery in software defined networks: Threats, taxonomy, and state-of-the-art,” *IEEE Communications Surveys Tutorials*, vol. 19, no. 1, pp. 303–324, 2017.
(Cited on page 13.)
- [79] T. A. V. Sattolo, S. Macwan, M. J. Vezina, and A. Matrawy, “Classifying poisoning attacks in software defined networking,” in *Proceedings of the 2019 IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE ’19)*. IEEE, 2019, pp. 59–64.
(Cited on page 13.)
- [80] C. Yoon, S. Lee, H. Kang, T. Park, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, “Flow wars: Systemizing the attack surface and defenses in software-defined networks,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3514–3530, Dec. 2017.
(Cited on pages 13, 66, 96, and 128.)
- [81] B. E. Ujcich, U. Thakore, and W. H. Sanders, “ATTAIN: An attack injection framework for software-defined networking,” in *Proceedings of the 2017 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’17)*. IEEE, June 2017.
(Cited on pages 15, 24, and 64.)
- [82] POX, “POX,” Aug. 2018. [Online]. Available: <https://github.com/noxrepo/pox>
(Cited on pages 16 and 141.)
- [83] D. Erickson, “The Beacon OpenFlow controller,” in *Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’13)*. ACM, 2013, pp. 13–18.
(Cited on page 16.)
- [84] A. Alimohammadifar, S. Majumdar, T. Madi, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debabi, “Stealthy probing-based verification (SPV): An active approach to defending software defined networks against topology poisoning attacks,” in *Computer Security*, J. Lopez, J. Zhou, and M. Soriano, Eds. Cham: Springer International Publishing, 2018, pp. 463–484.
(Cited on pages 21 and 23.)
- [85] S. Scott-Hayward and T. Arumugam, “OFMTL-SEC: State-based security for software defined networks,” in *Proceedings of the 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN ’18)*. IEEE, Nov. 2018, pp. 1–7.
(Cited on pages 21 and 23.)
- [86] C. Röpke and T. Holz, “Preventing malicious SDN applications from hiding adverse network manipulations,” in *Proceedings of the 2018 ACM SIGCOMM Workshop on Security in Softwarized Networks: Prospects and Challenges (SecSoN ’18)*. ACM, 2018, pp. 40–45.
(Cited on pages 21 and 22.)

- [87] D. Tatang, F. Quinkert, J. Frank, C. Röpke, and T. Holz, “SDN-Guard: Protecting SDN controllers against SDN rootkits,” in *Proceedings of the 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN ’17)*. IEEE, Nov. 2017, pp. 297–302. (Cited on pages 21 and 22.)
- [88] A. Shaghaghi, M. A. Kaafar, and S. Jha, “WedgeTail: An intrusion prevention system for the data plane of software defined networks,” in *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (AsiaCCS ’17)*. ACM, 2017, pp. 849–861. (Cited on pages 21 and 22.)
- [89] D. Smyth, S. McSweeney, D. O’Shea, and V. Cionca, “Detecting link fabrication attacks in software-defined networks,” in *Proceedings of the 2017 IEEE International Conference on Computer Communication and Networks (ICCCN ’17)*. IEEE, July 2017, pp. 1–8. (Cited on pages 21 and 23.)
- [90] B. Chandrasekaran, B. Tschaen, and T. Benson, “Isolating and tolerating SDN application failures with LegoSDN,” in *Proceedings of the 2016 ACM SIGCOMM Symposium on SDN Research (SOSR ’16)*. ACM, 2016, pp. 7:1–7:12. (Cited on pages 21 and 23.)
- [91] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, “SDNsec: Forwarding accountability for the SDN data plane,” in *Proceedings of the 2016 IEEE International Conference on Computer Communication and Networks (ICCCN ’16)*. IEEE, 2016. (Cited on pages 21, 23, and 29.)
- [92] K. Mahajan, R. Poddar, M. Dhawan, and V. Mann, “JURY: Validating controller actions in software-defined networks,” in *Proceedings of the 2016 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’16)*. IEEE, June 2016, pp. 109–120. (Cited on pages 21 and 22.)
- [93] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, “Sphinx: Detecting security attacks in software-defined networks,” in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS ’15)*. Internet Society, Feb. 2015. (Cited on pages 21, 23, 52, 66, 70, 95, 96, 99, 101, 112, 119, and 127.)
- [94] A. Kamisiński and C. Fung, “FlowMon: Detecting malicious switches in software-defined networks,” in *Proceedings of the 2015 ACM Workshop on Automated Decision Making for Active Cyber Defense (SafeConfig ’15)*. ACM, 2015, pp. 39–45. (Cited on pages 21 and 22.)
- [95] H. Wang, L. Xu, and G. Gu, “FloodGuard: a DoS attack prevention extension in software-defined networks,” in *Proceedings of the 2015 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’15)*. IEEE, 2015, pp. 239–250. (Cited on pages 21 and 24.)
- [96] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, “VeriFlow: Verifying network-wide invariants in real time,” *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, Sep. 2012.

(Cited on pages 21, 23, 64, 70, 97, and 128.)

- [97] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real time network policy checking using header space analysis,” in *Proceedings of the 2013 USENIX Symposium on Networked Systems Design and Implementation (NSDI ’13)*. USENIX Association, 2013, pp. 99–111.

(Cited on pages 21, 23, 64, and 128.)

- [98] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, “Onix: A distributed control platform for large-scale production networks,” in *Proceedings of the 2010 USENIX Symposium on Operating Systems Design and Implementation (OSDI ’10)*. USENIX Association, 2010.

(Cited on page 22.)

- [99] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, “DevoFlow: Scaling flow management for high-performance networks,” in *Proceedings of the 2011 ACM Special Interest Group on Data Communication Conference (SIGCOMM ’11)*. ACM, 2011, pp. 254–265.

(Cited on page 22.)

- [100] A. Tootoonchian and Y. Ganjali, “HyperFlow: A distributed control plane for OpenFlow,” in *Proceedings of the 2010 USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN ’10)*. USENIX Association, 2010.

(Cited on page 22.)

- [101] F. Botelho, A. Bessani, F. M. V. Ramos, and P. Ferreira, “On the design of practical fault-tolerant SDN controllers,” in *Proceedings of the 2014 IEEE European Workshop on Software Defined Networks (EWSDN ’14)*. IEEE, 2014.

(Cited on pages 22, 30, 32, and 36.)

- [102] N. Katta, H. Zhang, M. Freedman, and J. Rexford, “Ravana: Controller fault-tolerance in software-defined networking,” in *Proceedings of the 2015 ACM SIGCOMM Symposium on SDN Research (SOSR ’15)*. ACM, 2015.

(Cited on pages 22, 30, 32, and 36.)

- [103] P. Kazemian, G. Varghese, and N. McKeown, “Header space analysis: Static checking for networks,” in *Proceedings of the 2012 USENIX Symposium on Networked Systems Design and Implementation (NSDI ’12)*. USENIX Association, 2012.

(Cited on pages 23, 115, and 128.)

- [104] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, “Debugging the data plane with Anteater,” in *Proceedings of the 2011 ACM Special Interest Group on Data Communication Conference (SIGCOMM ’11)*. ACM, 2011.

(Cited on pages 24 and 128.)

- [105] M. Canini, D. Venzano, P. Perešini, D. Kostić, and J. Rexford, “A NICE way to test OpenFlow applications,” in *Proceedings of the 2012 USENIX Symposium on Networked Systems Design and Implementation (NSDI ’12)*. USENIX Association, 2012, pp. 127–140.

(Cited on pages 24, 64, 66, 70, 95, 97, and 128.)

- [106] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, “VeriCon: Towards verifying controller programs in software-defined networks,” in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. ACM, 2014.
(Cited on pages 24 and 128.)
- [107] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev, “SDNRacer: Detecting concurrency violations in software-defined networks,” in *Proceedings of the 2015 ACM SIGCOMM Symposium on SDN Research (SOSR ’15)*. ACM, 2015.
(Cited on pages 24 and 128.)
- [108] X. S. Sun, A. Agarwal, and T. S. E. Ng, “Controlling race conditions in OpenFlow to accelerate application verification and packet forwarding,” *IEEE Transactions on Network and Service Management*, vol. 12, no. 2, pp. 263–277, 2015.
(Cited on page 24.)
- [109] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, “Attacking the brain: Races in the SDN control plane,” in *Proceedings of the 2017 USENIX Security Symposium (Security ’17)*. USENIX Association, 2017.
(Cited on pages 24, 67, 70, 96, and 128.)
- [110] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, “Troubleshooting blackbox SDN control software with minimal causal sequences,” in *Proceedings of the 2014 ACM Special Interest Group on Data Communication Conference (SIGCOMM ’14)*. ACM, 2014.
(Cited on pages 24, 66, 97, and 128.)
- [111] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowrya, and S. Fahmy, “BEADS: Automated attack discovery in OpenFlow-based SDN systems,” in *Proceedings of the 2017 International Symposium on Research in Attacks, Intrusions and Defenses (RAID ’17)*, 2017.
(Cited on pages 24, 64, 67, 70, and 97.)
- [112] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, “DELTA: A security assessment framework for software-defined networks,” in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS ’17)*. Internet Society, Feb. 2017.
(Cited on pages 24, 40, 64, 66, 67, 70, and 97.)
- [113] C. Lee and S. Shin, “SHIELD: An automated framework for static analysis of SDN applications,” in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security ’16)*. ACM, 2016, pp. 29–34.
(Cited on pages 25, 52, 63, 67, and 96.)
- [114] C. Lee, C. Yoon, S. Shin, and S. K. Cha, “INDAGO: A new framework for detecting malicious SDN applications,” in *Proceedings of the 2018 IEEE International Conference on Network Protocols (ICNP ’18)*. IEEE, Sep. 2018.

(Cited on pages 25, 67, 75, 96, 127, and 131.)

- [115] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker, “Frenetic: A high-level language for OpenFlow networks,” in *Proceedings of the 2010 ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO '10)*. ACM, 2010, pp. 6:1–6:6.
(Cited on pages 25, 63, and 71.)
- [116] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing software defined networks,” in *Proceedings of the 2013 USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, 2013, pp. 1–13.
(Cited on pages 25, 63, and 71.)
- [117] C. Monsanto, N. Foster, R. Harrison, and D. Walker, “A compiler and run-time system for network programming languages,” in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 217–230.
(Cited on pages 25, 63, and 71.)
- [118] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, “Merlin: A language for provisioning network resources,” in *Proceedings of the 2014 ACM Conference on Emerging Networking Experiment and Technologies (CoNEXT '14)*. ACM, 2014, pp. 213–226.
(Cited on page 25.)
- [119] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, “NetKAT: Semantic foundations for networks,” in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 113–126.
(Cited on pages 25, 63, and 71.)
- [120] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, “Tierless programming and reasoning for software-defined networks,” in *Proceedings of the 2014 USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, Apr. 2014, pp. 519–531.
(Cited on page 25.)
- [121] L. Ryzhyk, N. Bjørner, M. Canini, J.-B. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese, “Correct by construction networks using stepwise refinement,” in *Proceedings of the 2017 USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. USENIX Association, Mar. 2017.
(Cited on pages 25 and 128.)
- [122] B. E. Ujcich, A. Miller, A. Bates, and W. H. Sanders, “Towards an accountable software-defined networking architecture,” in *Proceedings of the 2017 IEEE International Conference on Network Softwarization (NetSoft '17)*. IEEE, July 2017.
(Cited on pages 25, 64, and 127.)
- [123] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, “OFRewind: Enabling record and replay troubleshooting for networks,” in *Proceedings of the 2011 USENIX Annual Technical Conference (ATC '11)*. USENIX Association, 2011.

(Cited on pages 25 and 128.)

- [124] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, “I know what your packet did last hop: Using packet histories to troubleshoot networks,” in *Proceedings of the 2014 USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. USENIX Association, Apr. 2014, pp. 71–85.
(Cited on page 25.)
- [125] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, “Where is the debugger for my software-defined network?” in *Proceedings of the 2012 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '12)*. ACM, 2012.
(Cited on pages 25 and 128.)
- [126] R. May, A. El-Hassany, L. Vanbever, and M. Vechev, “BigBug: Practical concurrency analysis for SDN,” in *Proceedings of the 2017 ACM SIGCOMM Symposium on SDN Research (SOSR '17)*. ACM, 2017.
(Cited on pages 25 and 128.)
- [127] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, “NetzText: Query-guided summarization of network forwarding behaviors,” in *Proceedings of the 2018 USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*. USENIX Association, Apr. 2018.
(Cited on pages 25 and 128.)
- [128] X. Li, Y. Yu, K. Bu, Y. Chen, J. Yang, and R. Quan, “Thinking inside the box: Differential fault localization for SDN control plane,” in *Proceedings of the 2019 IFIP/IEEE International Symposium on Integrated Network Management (IM '19)*. IEEE, April 2019.
(Cited on pages 25 and 128.)
- [129] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, “Towards fine-grained network security forensics and diagnosis in the SDN era,” in *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS '18)*. ACM, 2018.
(Cited on pages 26, 67, 70, 79, 97, 100, 103, 104, 112, 114, 121, 122, and 127.)
- [130] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, “Answering why-not queries in software-defined networks with negative provenance,” in *Proceedings of the 2013 ACM Workshop on Hot Topics in Networks (HotNets '13)*. ACM, 2013.
(Cited on pages 26, 100, and 127.)
- [131] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, “Diagnosing missing events in distributed systems with negative provenance,” in *Proceedings of the 2014 ACM Special Interest Group on Data Communication Conference (SIGCOMM '14)*. ACM, 2014, pp. 383–394.
(Cited on pages 26, 64, 97, 100, 105, and 127.)
- [132] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, “Automated network repair with meta provenance,” in *Proceedings of the 2015 ACM Workshop on Hot Topics in Networks (HotNets '15)*. ACM, 2015, pp. 26:1–26:7.
(Cited on pages 26, 64, 97, 105, and 127.)

- [133] A. R. Yumerefendi and J. S. Chase, “Trust but verify: Accountability for network services,” in *Proceedings of the 2004 ACM SIGOPS European Workshop*, 2004.
(Cited on page 28.)
- [134] A. R. Yumerefendi and J. S. Chase, “The role of accountability in dependable distributed systems,” in *Proceedings of the 2005 USENIX Workshop on Hot Topics in System Dependability (HotDep ’05)*. USENIX Association, 2005.
(Cited on page 28.)
- [135] J. L. Mashaw, “Accountability and institutional design: Some thoughts on the grammar of governance,” in *Public Accountability: Designs, Dilemmas, and Experience*, M. W. Dowdle, Ed. Cambridge University Press, 2006, pp. 115–156.
(Cited on pages 28 and 29.)
- [136] A. Haeberlen, “A case for the accountable cloud,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 52–57, Apr. 2010.
(Cited on pages 30 and 35.)
- [137] Open Networking Foundation, “OF-CONFIG 1.2,” 2014. [Online]. Available: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf>
(Cited on page 31.)
- [138] P. Buneman, S. Khanna, and W. C. Tan, “Why and where: A characterization of data provenance,” in *Proceedings of the 2001 International Conference on Database Theory (ICDT ’01)*, 2001.
(Cited on page 31.)
- [139] Y. L. Simmhan, B. Plale, and D. Gannon, “A survey of data provenance in e-science,” *ACM SIGMOD Record*, vol. 34, no. 3, Sep. 2005.
(Cited on page 31.)
- [140] A. Haeberlen, P. Kouznetsov, and P. Druschel, “PeerReview: Practical accountability for distributed systems,” in *Proceedings of the 2007 ACM Symposium on Operating Systems Principles (SOSP ’07)*. ACM, 2007.
(Cited on page 31.)
- [141] A. Dwaraki, S. Seetharaman, S. Natarajan, and T. Wolf, “GitFlow: Flow revision management for software-defined networks,” in *Proceedings of the 2015 ACM SIGCOMM Symposium on SDN Research (SOSR ’15)*. ACM, 2015, pp. 6:1–6:6.
(Cited on pages 31, 36, 64, 100, and 127.)
- [142] S. A. Crosby and D. S. Wallach, “Efficient data structures for tamper-evident logging,” in *Proceedings of the 2009 USENIX Security Symposium (Security ’09)*. USENIX Association, 2009.
(Cited on page 31.)

- [143] M. Castro and B. Liskov, “Practical Byzantine fault tolerance,” in *Proceedings of the 1999 USENIX Symposium on Operating Systems Design and Implementation (OSDI ’99)*. USENIX Association, 1999.
(Cited on page 32.)
- [144] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, “Secure network provenance,” in *Proceedings of the 2011 ACM Symposium on Operating Systems Principles (SOSP ’11)*. ACM, 2011, pp. 295–310.
(Cited on pages 32 and 64.)
- [145] L. Jacquin, A. L. Shaw, and C. Dalton, “Towards trusted software-defined networks using a hardware-based integrity measurement architecture,” in *Proceedings of the 2015 IEEE International Conference on Network Softwarization (NetSoft ’15)*. IEEE, 2015.
(Cited on pages 32 and 36.)
- [146] Council of the European Union, “Regulation (EU) 2015/2120,” in *Official Journal of the European Union*, vol. L 310, 2015.
(Cited on page 33.)
- [147] F. B. Schneider, “Accountability for perfection,” *IEEE Security and Privacy*, vol. 7, no. 2, Mar. 2009.
(Cited on page 33.)
- [148] C. E. Landwehr, “A national goal for cyberspace: Create an open, accountable Internet,” *IEEE Security and Privacy*, vol. 7, no. 3, May 2009.
(Cited on page 33.)
- [149] Ethereum Project, “Ethereum white paper: A next-generation smart contract and decentralized application platform,” (Accessed 7 Jan. 2017). [Online]. Available: <https://github.com/ethereum/wiki/wiki/White-Paper>
(Cited on pages 33 and 35.)
- [150] Y. Zhang, N. Beheshti, and R. Manghirmalani, “NetRevert: Rollback recovery in SDN,” in *Proceedings of the 2014 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’14)*. ACM, 2014.
(Cited on pages 34 and 36.)
- [151] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, “Towards an elastic distributed SDN controller,” in *Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’13)*. ACM, 2013.
(Cited on pages 34 and 36.)
- [152] Hewlett-Packard Enterprise, “HPE SDN app store,” (Accessed 7 Jan. 2017). [Online]. Available: <https://marketplace.saas.hpe.com/sdn>
(Cited on pages 34 and 65.)

- [153] A. Bates, K. R. B. Butler, and T. Moyer, “Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs,” in *Proceedings of the 2015 USENIX International Workshop on the Theory and Practice of Provenance (TaPP ’15)*. USENIX Association, 2015.
(Cited on page 36.)
- [154] International Data Corporation, “SDN market to experience strong growth over next several years, according to IDC,” <https://www.idc.com/getdoc.jsp?containerId=prUS41005016>, Feb. 2016.
(Cited on page 37.)
- [155] S. Lee, C. Yoon, and S. Shin, “The smaller, the shrewder: A simple malicious application can kill an entire SDN environment,” in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security ’16)*. ACM, 2016, pp. 23–28.
(Cited on pages 38, 42, and 63.)
- [156] M. C. Dacier, H. König, R. Cwalinski, F. Kargl, and S. Dietrich, “Security challenges and opportunities of software-defined networking,” *IEEE Security & Privacy*, vol. 15, no. 2, pp. 96–100, March 2017.
(Cited on pages 38, 42, and 63.)
- [157] N. Hardy, “The confused deputy: (Or why capabilities might have been invented),” *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, Oct. 1988.
(Cited on page 38.)
- [158] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan 2003.
(Cited on pages 38, 41, and 42.)
- [159] Open Networking Foundation, “ONOS in action,” 2018. [Online]. Available: <https://onosproject.org/in-action/>
(Cited on pages 40, 46, 68, and 101.)
- [160] Open Networking Foundation, “Security advisories: ONOS,” 2018. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/Security+advisories>
(Cited on page 40.)
- [161] T. Pasquier, J. Singh, D. Eysers, and J. Bacon, “CamFlow: Managed data-sharing for cloud services,” *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, 2017.
(Cited on pages 41, 53, and 61.)
- [162] K. J. Biba, “Integrity considerations for secure computer systems,” MITRE Corporation, Tech. Rep. MTR-3153, June 1975.
(Cited on pages 41 and 62.)

- [163] A. C. Myers and B. Liskov, “A decentralized model for information flow control,” in *Proceedings of the 1997 ACM Symposium on Operating Systems Principles (SOSP ’97)*. ACM, 1997, pp. 129–142.
(Cited on pages 41 and 53.)
- [164] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner, “Corybantic: Towards the modular composition of SDN control programs,” in *Proceedings of the 2013 ACM Workshop on Hot Topics in Networks (HotNets ’13)*. ACM, 2013, pp. 1:1–1:7.
(Cited on page 42.)
- [165] R. J. Lipton and L. Snyder, “A linear time algorithm for deciding subject security,” *Journal of the ACM*, vol. 24, no. 3, pp. 455–464, July 1977.
(Cited on page 43.)
- [166] Open Networking Foundation, “GitHub – opennetworkinglab/onos at 1.10.0,” 2018. [Online]. Available: <https://github.com/opennetworkinglab/onos/tree/1.10.0>
(Cited on pages 46 and 137.)
- [167] D. van Bruggen, “JavaParser: For parsing Java code,” 2018. [Online]. Available: <http://javaparser.org/>
(Cited on page 50.)
- [168] Flick Team, “C abstract syntax tree (CAST) representation,” 2018. [Online]. Available: <http://www.cs.utah.edu/flux/flick/current/doc/guts/gutsch6.html>
(Cited on page 50.)
- [169] Python Software Foundation, “ast—abstract syntax trees,” 2018. [Online]. Available: <https://docs.python.org/3/library/ast.html>
(Cited on page 50.)
- [170] D. J. Tian, A. Bates, K. R. Butler, and R. Rangaswami, “ProvUSB: Block-level provenance-based data protection for USB storage devices,” in *Proceedings of the 2016 ACM Conference on Computer and Communications Security (CCS ’16)*. ACM, 2016, pp. 242–253.
(Cited on page 53.)
- [171] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao, “Chimera: A virtual data system for representing, querying, and automating data derivation,” in *Proceedings of the 2002 Scientific and Statistical Database Management Conference (SSDBM ’02)*, July 2002.
(Cited on page 54.)
- [172] J. Widom, “Trio: A system for integrated management of data, accuracy, and lineage,” Stanford InfoLab, Tech. Rep. 2004-40, Aug. 2004.
(Cited on page 54.)
- [173] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, “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)*. ACM, June 2005.
(Cited on page 54.)

- [174] B. Glavic and G. Alonso, “Perm: Processing provenance and data on the same data model through query rewriting,” in *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE ’09)*, Mar. 2009.
(Cited on page 54.)
- [175] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic, “A generic provenance middleware for database queries, updates, and transactions,” in *Proceedings of the 2014 USENIX International Workshop on the Theory and Practice of Provenance (TaPP ’14)*. USENIX Association, June 2014.
(Cited on page 54.)
- [176] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, “Hi-Fi: Collecting high-fidelity whole-system provenance,” in *Proceedings of the 2012 Annual Computer Security Applications Conference (ACSAC ’12)*, 2012, pp. 259–268.
(Cited on pages 54 and 128.)
- [177] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, “Provenance-aware storage systems,” in *Proceedings of the 2006 USENIX Annual Technical Conference (ATC ’06)*. USENIX Association, 2006.
(Cited on page 54.)
- [178] S. Ma, X. Zhang, and D. Xu, “ProTracer: Towards practical provenance tracing by alternating between logging and tainting,” in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS ’16)*. Internet Society, Feb. 2016.
(Cited on pages 54 and 128.)
- [179] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, “Trustworthy whole-system provenance for the Linux kernel,” in *Proceedings of the 2015 USENIX Security Symposium (Security ’15)*. USENIX Association, 2015, pp. 319–334.
(Cited on pages 54 and 128.)
- [180] M. Backes, S. Bugiel, and S. Gerling, “Scippa: System-centric IPC provenance on Android,” in *Proceedings of the 2014 Annual Computer Security Applications Conference (ACSAC ’14)*, 2014, pp. 36–45.
(Cited on page 54.)
- [181] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, “Quire: Lightweight provenance for smart phone operating systems,” in *Proceedings of the 2011 USENIX Security Symposium (Security ’11)*. USENIX Association, 2011.
(Cited on pages 54 and 62.)
- [182] K. H. Lee, X. Zhang, and D. Xu, “High accuracy attack provenance via binary-based execution partition,” in *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS ’13)*. Internet Society, Feb. 2013.
(Cited on pages 54 and 128.)

- [183] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, “MPI: Multiple perspective attack investigation with semantic aware execution partitioning,” in *Proceedings of the 2017 USENIX Security Symposium (Security ’17)*. USENIX Association, Aug. 2017.
(Cited on pages 54 and 128.)
- [184] L. Moreau and P. Groth, *Provenance: An introduction to PROV*, ser. Synthesis Lectures on the Semantic Web: Theory and Technology, J. Hendler and Y. Ding, Eds. Morgan & Claypool Publishers, 2013, vol. 3, no. 4.
(Cited on pages 54, 106, and 139.)
- [185] B. E. Ujcich, A. Bates, and W. H. Sanders, “A provenance model for the European Union General Data Protection Regulation,” in *Proceedings of the 2018 International Provenance and Annotation Workshop (IPAW ’18)*, 2018.
(Cited on page 54.)
- [186] P. Missier, K. Belhajjame, and J. Cheney, “The W3C PROV family of specifications for modelling provenance metadata,” in *Proceedings of the 2013 ACM International Conference on Extending Database Technology (EDBT ’13)*, 2013, pp. 773–776.
(Cited on pages 54 and 106.)
- [187] A. Gehani and D. Tariq, “SPADE: Support for provenance auditing in distributed environments,” in *Proceedings of the 2012 ACM/IFIP/USENIX International Conference on Middleware (Middleware ’12)*. Springer-Verlag New York, 2012, pp. 101–120.
(Cited on pages 54 and 128.)
- [188] A. Bates, D. J. Pohly, and K. R. B. Butler, “Secure and trustworthy provenance collection for digital forensics,” in *Digital Fingerprinting*, C. Wang, R. M. Gerdes, Y. Guan, and S. K. Kasera, Eds. Springer New York, 2016, pp. 141–176.
(Cited on page 54.)
- [189] D. van Heesch, “Doxygen: Generate documentation from source code,” 2018. [Online]. Available: <http://www.stack.nl/~dimitri/doxygen/>
(Cited on page 57.)
- [190] B. Naveh, “JGraphT,” 2018. [Online]. Available: <https://jgrapht.org/>
(Cited on pages 57, 85, and 120.)
- [191] OpenDaylight Project, “OpenDaylight documentation: Authentication, authorization and accounting (AAA) services,” 2018. [Online]. Available: <https://docs.opendaylight.org/en/stable-boron/user-guide/authentication-and-authorization-services.html>
(Cited on page 60.)
- [192] Project Floodlight, “Floodlight controller: MemoryStorageSource (dev),” 2018. [Online]. Available: <https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343633/MemoryStorageSource+Dev>
(Cited on page 60.)

- [193] A. Banerjee and D. A. Naumann, “Stack-based access control and secure information flow,” *Journal of Functional Programming*, vol. 15, no. 2, pp. 131–177, Mar. 2005.
(Cited on page 61.)
- [194] A. Banerjee and D. A. Naumann, “History-based access control and secure information flow,” in *Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS ’04)*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 27–48.
(Cited on page 61.)
- [195] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, “Jif: Java + information flow,” 2018. [Online]. Available: <http://www.cs.cornell.edu/jif/>
(Cited on page 61.)
- [196] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” in *Proceedings of the 2010 USENIX Symposium on Operating Systems Design and Implementation (OSDI ’10)*. USENIX Association, 2010, pp. 393–407.
(Cited on pages 62 and 96.)
- [197] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission re-delegation: Attacks and defenses,” in *Proceedings of the 2011 USENIX Security Symposium (Security ’11)*. USENIX Association, 2011.
(Cited on page 62.)
- [198] A. Nadkarni, B. Andow, W. Enck, and S. Jha, “Practical DIFC enforcement on Android,” in *Proceedings of the 2016 USENIX Security Symposium (Security ’16)*. USENIX Association, 2016, pp. 1119–1136.
(Cited on page 62.)
- [199] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian, “Run-time monitoring and formal analysis of information flows in Chromium,” in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS ’15)*. Internet Society, Feb. 2015.
(Cited on page 62.)
- [200] T. Fraser, “LOMAC: Low water-mark integrity protection for COTS environments,” in *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P ’00)*. IEEE, May 2000, pp. 230–245.
(Cited on page 62.)
- [201] A. Voellmy, H. Kim, and N. Feamster, “Procera: A language for high-level reactive network control,” in *Proceedings of the 2012 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN ’12)*, ACM. ACM, 2012, pp. 43–48.
(Cited on pages 63 and 71.)
- [202] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, “The good, the bad, and the differences: Better network diagnostics with differential provenance,” in *Proceedings of the 2016 ACM Special Interest Group on Data Communication Conference (SIGCOMM ’16)*. ACM, 2016, pp. 115–128.

(Cited on pages 64, 97, 105, and 127.)

- [203] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, “Cross-checking semantic correctness: The case of finding file system bugs,” in *Proceedings of the 2015 ACM Symposium on Operating Systems Principles (SOSP ’15)*. ACM, 2015.
(Cited on page 66.)
- [204] D. Hovemeyer and W. Pugh, “Finding more null pointer bugs, but not too many,” in *Proceedings of the 2007 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE ’07)*. ACM, 2007.
(Cited on page 66.)
- [205] PMD, “PMD: An extensible cross-language static code analyzer,” 2019. [Online]. Available: <https://pmd.github.io/>
(Cited on page 66.)
- [206] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: Using static analysis to find bugs in the real world,” *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.
(Cited on page 66.)
- [207] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, “Improving integer security for systems with KINT,” in *Proceedings of the 2012 USENIX Symposium on Operating Systems Design and Implementation (OSDI ’12)*. USENIX Association, 2012.
(Cited on page 66.)
- [208] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *Proceedings of the 2013 ACM International Conference on Software Engineering (ICSE ’13)*. ACM, 2013.
(Cited on page 66.)
- [209] J. Toman and D. Grossman, “Taming the static analysis beast,” in *Proceedings of the 2017 Summit on Advances in Programming Languages (SNAPL ’17)*, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, pp. 18:1–18:14.
(Cited on pages 66 and 81.)
- [210] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages*, vol. 9, no. 3, pp. 319–349, July 1987.
(Cited on page 71.)
- [211] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P ’14)*. IEEE, 2014.
(Cited on pages 71 and 97.)
- [212] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 1990.
(Cited on page 75.)

- [213] G. Jeh and J. Widom, “SimRank: A measure of structural-context similarity,” in *Proceedings of the 2002 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’02)*. ACM, 2002.
(Cited on pages 75 and 147.)
- [214] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: Statically vetting Android apps for component hijacking vulnerabilities,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS ’12)*. ACM, 2012.
(Cited on pages 79, 96, and 97.)
- [215] V. B. Livshits and M. S. Lam, “Finding security vulnerabilities in Java applications with static analysis,” in *Proceedings of the 2005 USENIX Security Symposium (Security ’05)*. USENIX Association, 2005.
(Cited on pages 83 and 97.)
- [216] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “AEG: Automatic exploit generation,” in *Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS ’11)*. Internet Society, Feb. 2011.
(Cited on pages 83 and 96.)
- [217] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, Nov. 2011.
(Cited on page 85.)
- [218] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot: A Java byte-code optimization framework,” in *Proceedings of the 2010 IBM Center for Advanced Studies Conference (CASCON ’10)*, 2010.
(Cited on page 85.)
- [219] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, July 1987.
(Cited on page 85.)
- [220] Y. Shoshitaishvili, R. F. Wang, A. Dutcher, C. Hauser, J. Grosen, C. Salls, N. Stephens, N. Rellini, C. Kruegel, and G. Vigna, “angr, a binary analysis framework,” <http://angr.io/>, 2017.
(Cited on page 85.)
- [221] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders,” in *Proceedings of the 2011 ACM International Conference on Software Engineering (ICSE ’11)*. ACM, 2011.
(Cited on page 85.)
- [222] P. Biondi, “Scapy: Packet crafting for Python2 and Python3,” 2019. [Online]. Available: <https://scapy.net/>
(Cited on page 86.)

- [223] Open Networking Foundation, “Virtual BNG,” 2019. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/Virtual+BNG>
(Cited on page 93.)
- [224] Open Networking Foundation, “Virtual network subsystem,” 2019. [Online]. Available: <https://wiki.onosproject.org/download/attachments/6357849/VirtualNetworkSubsystem.pdf>
(Cited on page 93.)
- [225] Open Networking Foundation, “Overlay VPNs and Gluon,” 2019. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/Overlay+VPNs+and+Gluon>
(Cited on page 94.)
- [226] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Oceau, and P. McDaniel, “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps,” in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. ACM, 2014.
(Cited on page 96.)
- [227] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Oceau, and P. McDaniel, “IccTA: Detecting inter-component privacy leaks in Android apps,” in *Proceedings of the 2015 ACM International Conference on Software Engineering (ICSE ’15)*. ACM, 2015.
(Cited on page 96.)
- [228] C. Qian, X. Luo, Y. Le, and G. Gu, “VulHunter: Toward discovering vulnerabilities in Android applications,” *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan. 2015.
(Cited on page 96.)
- [229] M. Zhang and H. Yin, “AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications,” in *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS ’14)*. Internet Society, Feb. 2014.
(Cited on page 96.)
- [230] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, “Semantics-aware Android malware classification using weighted contextual API dependency graphs,” in *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS ’14)*. ACM, 2014.
(Cited on page 96.)
- [231] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, “AppContext: Differentiating malicious and benign mobile app behaviors using context,” in *Proceedings of the 2015 ACM International Conference on Software Engineering (ICSE ’15)*. ACM, 2015.
(Cited on page 96.)
- [232] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, “VEX: Vetting browser extensions for security vulnerabilities,” in *Proceedings of the 2010 USENIX Security Symposium (Security ’10)*. USENIX Association, 2010.
(Cited on page 97.)

- [233] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, “Protecting browsers from extension vulnerabilities,” in *Proceedings of the 2010 Network and Distributed System Security Symposium (NDSS ’10)*. Internet Society, Feb. 2010.
(Cited on page 97.)
- [234] N. Carlini, A. P. Felt, and D. Wagner, “An evaluation of the Google Chrome extension security architecture,” in *Proceedings of the 2012 USENIX Security Symposium (Security ’12)*. USENIX Association, 2012.
(Cited on page 97.)
- [235] L. Liu, X. Zhang, G. Yan, and S. Chen, “Chrome extensions: Threat analysis and countermeasures,” in *Proceedings of the 2012 Network and Distributed System Security Symposium (NDSS ’12)*. Internet Society, Feb. 2012.
(Cited on page 97.)
- [236] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, “Static control-flow analysis of user-driven callbacks in Android applications,” in *Proceedings of the 2015 ACM International Conference on Software Engineering (ICSE ’15)*. ACM, 2015.
(Cited on page 97.)
- [237] M. Monperrus and M. Mezini, “Detecting missing method calls as violations of the majority rule,” *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 7:1–7:25, Mar. 2013.
(Cited on page 97.)
- [238] MITRE, “CVE-2018-12691: Time-of-check to time-of-use (TOCTOU) race condition in org.onosproject.acl.” [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=2018-12691>
(Cited on page 101.)
- [239] Y. Wu, A. Chen, and L. T. X. Phan, “Zeno: Diagnosing performance problems with temporal provenance,” in *Proceedings of the 2019 USENIX Symposium on Networked Systems Design and Implementation (NSDI ’19)*. USENIX Association, 2019.
(Cited on pages 105 and 128.)
- [240] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, “NoDoze: Combatting threat alert fatigue with automated provenance triage,” in *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS ’19)*. Internet Society, Feb. 2019.
(Cited on pages 117 and 128.)
- [241] Y. Xie, K.-K. Muniswamy-Reddy, D. D. E. Long, A. Amer, D. Feng, and Z. Tan, “Compressing provenance graphs,” in *Proceedings of the 2011 USENIX International Workshop on the Theory and Practice of Provenance (TaPP ’11)*. USENIX Association, June 2011.
(Cited on page 127.)
- [242] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, Y. Li, and D. D. E. Long, “Evaluation of a hybrid approach for efficient provenance storage,” *ACM Transactions on Storage*, vol. 9, no. 4, pp. 14:1–14:29, Nov. 2013.

(Cited on page 127.)

- [243] A. Chapman, H. Jagadish, and P. Ramanan, “Efficient provenance storage,” in *Proceedings of the 2008 ACM Special Interest Group on Management of Data Conference (SIGMOD ’08)*. ACM, 2008.
(Cited on page 127.)
- [244] K. H. Lee, X. Zhang, and D. Xu, “LogGC: Garbage collecting audit log,” in *Proceedings of the 2013 ACM Conference on Computer and Communications Security (CCS ’13)*. ACM, 2013.
(Cited on page 127.)
- [245] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, “Towards scalable cluster auditing through grammatical inference over provenance graphs,” in *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS ’18)*. Internet Society, Feb. 2018.
(Cited on page 127.)
- [246] A. Gehani, H. Kazmi, and H. Irshad, “Scaling SPADE to ‘big provenance,’” in *Proceedings of the 2016 USENIX International Workshop on the Theory and Practice of Provenance (TaPP ’16)*. USENIX Association, 2016.
(Cited on page 127.)
- [247] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long, “A hybrid approach for efficient provenance storage,” in *Proceedings of the 2012 ACM Conference on Information and Knowledge Management (CIKM ’12)*. ACM, 2012, pp. 1752–1756.
(Cited on page 127.)
- [248] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, “AIM-SDN: Attacking information mismanagement in SDN-datastores,” in *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS ’18)*. ACM, 2018.
(Cited on page 127.)
- [249] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, “Automated bug removal for software-defined networks,” in *Proceedings of the 2017 USENIX Symposium on Networked Systems Design and Implementation (NSDI ’17)*. USENIX Association, 2017.
(Cited on page 127.)
- [250] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, “Differential provenance: Better network diagnostics with reference events,” in *Proceedings of the 2015 ACM Workshop on Hot Topics in Networks (HotNets ’15)*. ACM, 2015.
(Cited on page 127.)
- [251] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, “Declarative networking: Language, execution and optimization,” in *Proceedings of the 2006 ACM Special Interest Group on Management of Data Conference (SIGMOD ’06)*. ACM, 2006.
(Cited on page 127.)

- [252] W. U. Hassan, M. A. Nouredine, P. Datta, and A. Bates, “OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis,” in *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS ’20)*. Internet Society, Feb. 2020.
(Cited on page 128.)
- [253] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, “Accurate, low cost and instrumentation-free security audit logging for Windows,” in *Proceedings of the 2015 Annual Computer Security Applications Conference (ACSAC ’15)*, 2015.
(Cited on page 128.)
- [254] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, “Kernel-supported cost-effective audit logging for causality tracking,” in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC ’18)*. USENIX Association, 2018.
(Cited on page 128.)
- [255] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, “Rain: Refinable attack investigation with on-demand inter-process information flow tracking,” in *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS ’17)*. ACM, 2017.
(Cited on page 128.)
- [256] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, and G. Ciocarlie, “MCI: Modeling-based causality inference in audit logging for attack investigation,” in *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS ’18)*. Internet Society, Feb. 2018.
(Cited on page 128.)
- [257] M. N. Hossain, S. Sheikhi, and R. Sekar, “Combating dependence explosion in forensic analysis using alternative tag propagation semantics,” in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P ’20)*. IEEE, 2020.
(Cited on page 128.)
- [258] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Intrusion recovery using selective re-execution,” in *Proceedings of the 2010 USENIX Symposium on Operating Systems Design and Implementation (OSDI ’10)*. USENIX Association, 2010.
(Cited on page 128.)
- [259] S. T. King and P. M. Chen, “Backtracking intrusions,” in *Proceedings of the 2003 ACM Symposium on Operating Systems Principles (SOSP ’03)*. ACM, 2003.
(Cited on page 128.)
- [260] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, “Enriching intrusion alerts through multi-host causality,” in *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS ’05)*. Internet Society, 2005.
(Cited on page 128.)
- [261] R. Hasan, R. Sion, and M. Winslett, “Preventing history forgery with secure provenance,” *ACM Transactions on Storage*, vol. 5, no. 4, pp. 12:1–12:43, Dec. 2009.

(Cited on page 128.)

- [262] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, “Layering in provenance systems,” in *Proceedings of the 2009 USENIX Annual Technical Conference (ATC ’09)*. USENIX Association, 2009.
(Cited on page 128.)
- [263] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, “HOLMES: Real-time APT detection through correlation of suspicious information flows,” in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (Se&P ’19)*. IEEE, 2019.
(Cited on page 128.)
- [264] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, “Fear and logging in the Internet of things,” in *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS ’18)*. Internet Society, Feb. 2018.
(Cited on page 128.)
- [265] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, “SLEUTH: Real-time attack scenario reconstruction from COTS audit data,” in *Proceedings of the 2017 USENIX Security Symposium (Security ’17)*. USENIX Association, 2017.
(Cited on page 128.)
- [266] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, “HERCULE: Attack story reconstruction via community discovery on correlated log graph,” in *Proceedings of the 2016 Annual Computer Security Applications Conference (ACSAC ’16)*, 2016.
(Cited on page 128.)
- [267] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, “Transparent web service auditing via network provenance functions,” in *Proceedings of the 2017 International World Wide Web Conference (WWW ’17)*, 2017.
(Cited on page 128.)
- [268] M. Stamatogiannakis, P. Groth, and H. Bos, “Looking inside the black-box: Capturing data provenance using dynamic instrumentation,” in *Proceedings of the 2015 International Provenance and Annotation Workshop (IPAW ’15)*, 2015.
(Cited on page 128.)
- [269] D. Tariq, M. Ali, and A. Gehani, “Towards automated collection of application-level data provenance,” in *Proceedings of the 2012 USENIX International Workshop on the Theory and Practice of Provenance (TaPP ’12)*. USENIX Association, 2012.
(Cited on page 128.)
- [270] E. Gessiou, V. Pappas, E. Athanasopoulos, A. D. Keromytis, and S. Ioannidis, “Towards a universal data provenance framework using dynamic instrumentation,” in *Information Security and Privacy Research*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds. Springer Berlin Heidelberg, 2012.
(Cited on page 128.)

- [271] A. Greenberg, “The untold story of NotPetya, the most devastating cyberattack in history,” Aug. 2018, *Wired*. [Online]. Available: <https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/>
(Cited on page 133.)
- [272] R. Sommer and V. Paxson, “Outside the closed world: On using machine learning for network intrusion detection,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P ’10)*, 2010, pp. 305–316.
(Cited on page 134.)
- [273] S. Kodeswaran, M. T. Arashloo, P. Tamma, and J. Rexford, “Tracking P4 program execution in the data plane,” in *Proceedings of the 2020 ACM SIGCOMM Symposium on SDN Research (SOSR ’20)*. ACM, 2020, pp. 117–122.
(Cited on page 134.)
- [274] M. V. Dumitru, D. Dumitrescu, and C. Raiciu, “Can we exploit buggy P4 programs?” in *Proceedings of the 2020 ACM SIGCOMM Symposium on SDN Research (SOSR ’20)*. ACM, 2020, pp. 62–68.
(Cited on page 134.)
- [275] Open Networking Foundation, “Enabling Security-Mode ONOS,” 2018. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/Enabling+Security-Mode+ONOS>
(Cited on page 137.)
- [276] Open Networking Foundation, “Creating security-mode compatible ONOS applications,” 2018. [Online]. Available: <https://wiki.onosproject.org/display/ONOS/Creating+Security-Mode+compatible+ONOS+applications>
(Cited on page 137.)
- [277] Hewlett-Packard Enterprise, “Software defined networking,” Aug. 2018. [Online]. Available: <https://www.hpe.com/us/en/networking/sdn.html>
(Cited on page 142.)