SECURITY ENHANCEMENT OF THE VISIBILITY AND PROGRAMMABILITY OF SOFTWARE-DEFINED NETWORKS

A Dissertation

by

LEI XU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Guofei Gu
Committee Members,	Alex Sprintson
	James Caverlee
	Radu Stoleru
Head of Department,	Dilma Da Silva

August 2019

Major Subject: Computer Engineering

ABSTRACT

Software-Defined Networking (SDN) is a new networking paradigm that centralizes the control logic from the data plane. Benefits from its centralized control plane (or SDN Controller), SDN intends to provide two key innovations, i.e., holistic network visibility and flexible network programmability, and thus to enable innovative network application scenarios ranging from campus network innovation to cloud network virtualization and data center network optimization. Unfortunately, the security issues and limitations of those two innovations are rarely explored, which put SDN-based infrastructures at risk.

In this thesis, we conduct in-depth security analysis upon SDN-provisioned network visibility and programmability. As a result, we locate several security issues and limitations that may impede current SDN to achieve its goals. First, network visibility depends on a reliable topology management service. However, we find that existing topology management services in SDN are vulnerable to network topology poisoning attacks, which thereby misleads topology-dependent services and applications. Second, programmability enables the concurrent execution of multiple apps/modules in SDN to efficiently process network events. However, we find that the concurrency of SDN is vulnerable to harmful race conditions, which can be exploited by state manipulation attacks and cause serious security and reliability issues. Finally, the current SDN visibility and programmability only cover network flow-level information, which is far from enough to secure the entire infrastructure in today's enterprise/cloud systems. It is because most of the recent cyber attacks involve many system-level malicious activities to attack system resources (e.g., a file hijacking by ransomware).

To tackle these problems, we propose new security solutions to significantly enhance existing SDN on its visibility and programmability with three major components, i.e., TOPOGUARD, CONGUARD, and SYSFLOW. TOPOGUARD works as a security extension on the SDN controller that secures the topology management by providing light-weighted, automatic, and real-time detection of topology poison attacks. CONGUARD works as a dynamic framework to effectively

detect and exploit those harmful race conditions in SDN controllers. SYSFLOW works as a unified programmable security framework to facilitate the enforcement of diverse security intents to secure both network and system resources by abstracting system level activities and security capabilities. We believe our experience and lessons are of great benefit to design and implement more secure SDN architecture.

ACKNOWLEDGMENTS

First of all, I am extremely grateful to my advisor, Prof. Guofei Gu, for his help and support during my Ph.D. study. Without his inspired advising, patient guidance, and continuous support, I cannot complete this dissertation. I would also like to thank the rest of my committee members, Prof. Alex Sprintson, Prof. James Caverlee, and Prof. Radu Stoleru, for their excellent comments and suggestions on my dissertation study.

I would like to express my gratitude to my mentors, Dr. Phillip Porras, and Dr. David Ott, for advising me during my internships in SRI International and VMware, respectively. I also want to thank my collaborators on finished and ongoing projects including Dr. Seungwon Shin, Dr. Richard Skowyra, Dr. Hamed Okhravi, Prof. Hongxin Hu, and Hongda. It was a pleasure to work with each of them.

I want to appreciate previous and current SUCCESS Lab members, Zhaoyan, Robert, Chao, Jialong, Haopei, Guangliang, Kevin, Abner, Yangyong, Patrick, Hao, Menghao, and Raj, for generous assistance on my research and life.

Last but not least, I take this opportunity to thank my parents and Jingjia. Without their enduring love and support, I could not have done this. I dedicate this dissertation to them.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis (or) dissertation committee consisting of Professor Guofei Gu, Professor Radu Stoleru and Professor James Caverlee of the Department of Computer Science & Engineering and Professor Alex Sprintson of the Department of Electrical & Computer Engineering.

The security analysis for Chapter 2 was conducted in part by Sungmin Hong of the Department of Electrical & Computer Engineering. Chapter 3 was completed by the student in collaboration with Professor Jeff Huang and Jialong Zhang of the Department of Computer Science & Engineering. The system implementation in Chapter 4 collaborated with Hao Jin of the Department of Computer Science & Engineering and Hongda Li of the School of Computing at the Clemson University.

Funding Sources

This work was made possible in part by National Science Foundation (NSF) under Grant no. 1617985, 1642129, 1740791, and 1700544, Air Force Office of Scientific Research (AFOSR) under FA-9550-13-1-0077 and a Google Faculty Research award.

Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the NSF, AFOSR and Google.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Modern networks, e.g., enterprise networks or data center networks, are posing more enormous management challenges due to their expanding size and complexity. In addition to best-effort packet forwarding, operators require to enforce more and more various network applications/services/protocols for different administrative requirements, including, but not limited to, elastic traffic routing, network virtualization, access control, and intrusion detection. However, legacy networking paradigms can hardly capture the pace to innovate new network applications, services, and protocols for two primary reasons: first, it is challenging to acquire a holistic network view and make effective network-wide decisions in legacy networking paradigms because the control plane resides in distributed and heterogeneous network devices, e.g., switches, router, and firewalls; second, it requires tedious and error-prone manual efforts to configure each device to enforce desired network-wide intents since the control plane is tightly coupled with the dedicated and proprietary data plane devices.

Recently, Software-Defined Networking (SDN) has emerged as a novel networking paradigm to innovate the ossified network infrastructure fundamentally. By decoupling the control logic from the data plane, SDN provides a logically centralized control plane (or SDN controller), the general data plane packet-processing model, and standard communication protocols, e.g., OpenFlow [1], to simplify the enforcement of various network management tasks. Upon the controller, SDN intends to achieve two key innovations, i.e., *flexible network programmability* and *holistic network visibility*, and thus to support operators to program innovative applications for different network management scenarios, such as traffic engineering to data center virtualization, fine-grained access control.

Flexible Network Programmability. Instead of individually configuring control logic into heterogeneous and vendor-specific data plane devices, SDN introduces a centralized control plane to management unified data plane devices with a general packet processing abstraction, i.e., "Match-Action" paradigm. By abstracting the control logic into the centralized control plane, SDN pro-

vides a strong innovation to enable developers to flexibly program network applications with different functionalities. The innovation is similar to smart-phone platforms (e.g., Android [2] and iOS [3]) that allow software engineers to develop mobile applications with flexible and creative functionalities. In order to empower this innovation, several network programming languages and frameworks in the SDN control plane have been proposed so far to further simplify the development of novel network functionalities, such as general-purpose SDN controllers [4, 5, 6, 7, 8] and domain-specific programming languages [9, 10, 11, 12, 13].

Holistic Network Visibility. In an SDN network, all data plane devices are managed by the centralized control plane via various control messages, such as flow rule modification messages and data plane configuration messages. The control plane can collect network status information from each data plane device by sending statistics query messages. Therefore, an SDN application running on the control plane naturally has the holistic visibility of all connected data plane devices. Based on holistic visibility, SDN can facilitate its applications to effectively and efficiently enforce custom network functions/algorithms. The innovation of holistic network visibility is highlighted in many applications, including, but not limited to efficient routing [14, 15, 16, 17], network-wide monitoring [18, 19, 20, 21], and security [22, 23, 24].

Despite the innovative benefits, the security issues and limitations of SDN-provisioned visibility and programmability are rarely investigated, which may threaten modern infrastructures, e.g., enterprise or cloud, that adopt SDN techniques. Motivated by the fact, in this thesis, we first perform in-depth security analysis on the network visibility and programmability provided by SDN. Consequently, we present three security issues and limitations (as detailed in Section 1.1) including: first, vulnerable topology management services in SDN significantly undermine network visibility; second, harmful race conditions seriously hazard SDN-provisioned network programmability; and third, the lack of system security visibility and programmability substantially limits SDN's adoption to ensure the security of modern cloud/enterprise infrastructures. To tackle the security issues and limitations, we propose our solutions (as detailed in Section 1.2) to fortify SDN with three goals: 1) enhance the security of network visibility provided by SDN; 2) enhance

the security of network programmability provided by SDN; and 3) enhance system security visibility and programmability to SDN.

1.1 Security Analysis on Network Visibility and Programmability in SDN

In this thesis, we perform in-depth security analysis upon network visibility and programmability provided by SDN in three aspects: first, we aim to investigate if the holistic network visibility provided by SDN is reliable; second, we want to examine if the flexible network programmability provided by SDN is secure; third, we intend to investigate if SDN-provisioned visibility and programmability are enough to address emerging security threats in modern infrastructures, e.g., cloud or enterprise. As a result, we locate three previously unknown or under-explored security issues and limitations as follows.

Firstly, network visibility in SDN depends on a reliable topology management service. However, from a systematic security analysis, we uncover new security loopholes existing in current topology management services, i.e., Host Tracking Service and Link Discovery Service, in SDN controllers. For the Host Track Service, there have few security restrictions on host migration. For the Link Discovery Service, it leverages Link Layer Discovery Protocol (LLDP) messages to discover switch links. However, the LLDP can be falsified or relayed in SDN networks. Furthermore, we introduce two Network Topology Poisoning Attacks, i.e., Host Location Hijacking Attack and Link Fabrication Attack. Upon the exploitation of the Host Tracking Service, an attacker can hijack the location of a network server to phish its service subscribers. By poisoning the Link Discovery Service, an adversary can inject falsified links (via fake LLDP injections or host-relayed LLDP) to create a black-hole route or launch a man-in-the-middle attack to eavesdrop/manipulate messages in the network. More details about the security issue of network visibility in SDN is covered in Chapter 2.

Secondly, network programmability in SDN enables application developers with the concurrent programming model to efficiently process network events in the large-volume and asynchronous nature. Based on systematic study on concurrency programming model in widely-used SDN controllers, we find that network programmability in SDN is vulnerable to concurrency vulnerabilities,

i.e., harmful race conditions, which can be exploited by the attackers to cause the denial of services (e.g., controller crash, core service disruption) and privacy leakage, etc. Based on the harmful race conditions, we present a new attack, namely State Manipulation Attack, against the security and reliability of the SDN control plane. We note that this attack is closely tied to the unique SDN semantics, which makes all popular SDN controllers (e.g., Floodlight [25], ONOS [26], and OpenDaylight [27]) vulnerable. We discuss more details about such vulnerabilities against network programmability in Chapter 3.

Finally, the current SDN-provisioned visibility and programmability only cover network flow level information, which is far from enough to secure the entire infrastructure in today's enterprise/cloud systems. The reason lies in that emerging cyber attacks tend to leverage elusive multi-stage attack strategies that involve many system-level malicious activities. Unfortunately, in many cases, existing SDN techniques fail to capture and prevent such advanced attacks due to it lacks system-level security visibility and programmability. For example, an SDN-based firewall can hardly prevent outgoing traffic associated with data ex-filtration attack if the leaked sensitive data is encrypted by the attacker. More details about confined visibility and programming in SDN is discussed in Chapter 4.

1.2 Solution Overview

As shown in Figure 1.1, in this thesis, we aim to provide a framework to significantly enhance the security of existing SDN on its innovative visibility and programmability with three key components, i.e., TOPOGUARD [28], CONGUARD [29] and SYSFLOW. TOPOGUARD is to enhance the security of network visibility by providing light-weighted, automatic, and real-time detection of network falsification attempts, i.e., topology poisoning attacks. CONGUARD is to enhance the security of network programmability in SDN by providing a dynamic race detection framework to detect and eliminate harmful race conditions. SYSFLOW is to enhance the SDN control plane with system security visibility and programmability to fortify modern cloud/enterprise infrastructures. We describe each component in detail as follows.

Firstly, to secure SDN-provisioned network visibility, we carefully analyze various possible

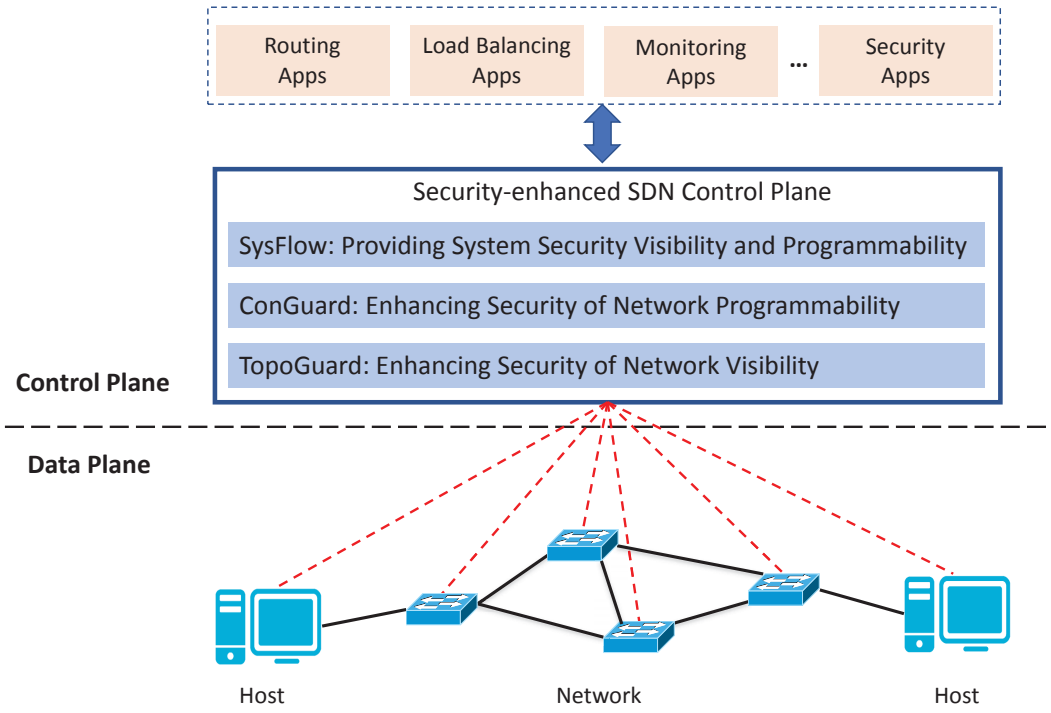


Figure 1.1: The overview of my thesis work

countermeasure strategies against topology poisoning attacks in SDN (i.e., host location hijacking attacks and link fabrication attacks). Based on the countermeasure analysis, we present TOPOGUARD, as a security extension on the SDN controller, to secure the topology management by providing light-weighted, automatic, and real-time detection of topology poison attacks. On the one hand, TOPOGUARD proposes to detect link fabrication attacks in two aspects: first, it utilizes a keyed-hash message authentication code (HMAC) as an authenticator during link discovery procedure to discover fake LLDP injections; second, it pinpoints host-relayed LLDP messages by leveraging behavior analysis to check the role of switch ports during LLDP propagation. On the other hand, TOPOGUARD proposes to detect host location hijacking attacks by verifying the legitimacy of Host Migration in with both pre-condition and post-condition. The pre-condition lies in

a port disconnection signal before a host migration happens. The post-condition is that the target host is not reachable in the previous location after a host migration happens. As shown in Chapter 2, the results show that TOPOGUARD is effective to defend against topology poisoning attacks by adding only minor overhead.

Secondly, to secure network programmability provided by SDN from harmful race conditions, we leverage race detection techniques to detect and patch those serious concurrency vulnerabilities in SDN control plane before their exploitation by attackers. In this work, we propose a dynamic framework, called CONGUARD, to effectively detect and validate those harmful race conditions in SDN controllers. In particular, we solve two research challenges in CONGUARD. First, detecting race conditions in the SDN control plane is generally undecidable due to lack of accurate modeling of the SDN semantics. Second, deciding if a race condition is harmful or not is difficult since they may be non-deterministic that only occur in rare scenarios with the specific input and schedule. To address the first challenge, we present a novel concurrency model of the SDN control plane in the form of happens-before semantics. Based on the SDN unique concurrency model, we can effectively locate race operations based on dynamic analysis. To address the second challenge, we develop a technique called *adversarial state racing* to detect harmful race conditions in the SDN control plane. Because there is no pre-defined order between the two race operations, we can hence actively control the scheduler (e.g., by inserting delays) to run an adversarial schedule, which forces one operation to execute after another. If we observe an erroneous state (e.g., an exception or a crash) in the adversarial schedule, we have found a harmful race condition. As shown in Chapter 3, CONGUARD effectively locates 15 harmful race conditions in the SDN control plane. Based on the discovery, we assist the developer in patching most of them.

Thirdly, to secure modern infrastructure from emerging advanced attacks, we propose to enhance SDN with system security visibility and programmability. For this purpose, we face two research questions. First, can we model system activities and security capabilities tightly coupled with low-level running system and hardware related details? Second, with the unified abstraction, can we provide an SDN-compatible framework to effectively and efficiently enforce holistic sys-

tem visibility and flexible programmability? To answer the first question, we introduce a novel flow-based model, namely system flow, to abstract system activities. Based on the system flow model, we introduce system flow rules that can be used to represent system security intents. To answer the second question, upon the flow-based system security abstraction, we propose a novel framework, namely SYSFLOW, to provide system security visibility and programmability. To be compatible with SDN, SYSFLOW embraces a two-layer design including SYSFLOW Data Plane and SYSFLOW Controller. The SYSFLOW Data Plane automatically enforces system flow rules to enable fine-grained responsive security actions, and dynamically update security intents (in the form of system flow rules) according to the change of contexts. The logically centralized SYSFLOW Controller acquires holistic visibility of security contexts from installed system flow rules in host systems and provides a unified programming abstraction to facilitate the flexible implementation and deployment of diverse SYSFLOW security intents based on system flows, even across the entire infrastructure. As shown in Chapter 4, SYSFLOW can effectively help SDN to defend against advanced cyber attacks with a minor performance overhead.

In all, we combine the three components into a generalized framework, which can serve as a robust framework to secure SDN architecture. The framework first leverages the functionality of TOPOGUARD component to inspect control messages from the network data plane (e.g., SDN switches), to detect and block spoofing attacks against network visibility, i.e., topology poisoning attempts. We consider this component can be extended to defend against more network-side attacks against the SDN control plane. Moreover, the framework utilizes CONGUARD component to instrument the SDN controller and its applications to generate execution traces and detect vulnerabilities, i.e., harmful race conditions, in an offline manner. The identified vulnerabilities can help developers to secure the SDN programmability by offline patching or online virtual patching. We can also extend this component to pinpoint and vet more vulnerable/malicious SDN applications. Besides, the framework uses SYSFLOW to extend a unified data plane into host systems and thus enhance SDN applications with system visibility and programmability to fortify modern infrastructures, e.g., cloud or enterprise. This component provides us more insights to abstract/complement

SDN with more security capabilities from other systems/components, e.g., virtual machines or containers. We will discuss more about the integration and extension of our proposed components in Chapter 5.

We organize the remainder of this thesis as follows. The following three chapters, Chapter 2, 3, 4, present the background, motivation, research problems, design, implementation, and evaluation results of the TOPOGUARD, CONGUARD, SYSFLOW, respectively. Moreover, in Chapter 5, we discuss our learned lessons and a generalized secure SDN architecture by integrating and extending of our proposed solutions. Finally, Chapter 6 concludes the entire thesis work and provides future work for this thesis.

2. TOPOGUARD: ENHANCING SECURITY OF NETWORK VISIBILITY IN SDN*

2.1 Introduction

Software-Defined Networking (SDN) has emerged as a new network paradigm to innovate the ossified network infrastructure by separating the control plane from the data plane (e.g., switches), as well as providing holistic network visibility. With the centralized holistic network visibility, many application scenarios have been studied and deployed since then, ranging from campus network innovation to cloud network virtualization and data center network optimization. However, the security of network visibility provided by SDN controllers is under-explored, which leaves a great room for adversaries to attack SDN networks.

In this thesis, we study the security of SDN-provisioned network visibility in terms of network topology management services/modules of the mainstream SDN controllers. As a fundamental building block for network visibility, the topology information is adopted to most controller core services and upper-layer apps, e.g., those related to packet routing, mobility tracking, network virtualization, and optimization. From the study, we identify several new vulnerabilities that an attacker can exploit to poison the network topology information in SDN networks. If such fundamental network topology information is poisoned, all the dependent network services will become immediately affected, causing catastrophic problems. For example, the routing services/apps inside the SDN controller can be manipulated to incur a black hole route or man-in-the-middle attack. In particular, we uncover new security loopholes existing in current Host Tracking Service and Link Discovery Service in SDN controllers. Furthermore, We introduce two Network Topology Poisoning Attacks, i.e., Host Location Hijacking Attack and Link Fabrication Attack. Upon the exploitation of the Host Tracking Service, an attacker can hijack the location of a network

*Reprinted with permission from "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures" by Lei Xu, Sunmin Hong, Haopei Wang and Guofei Gu, In Proceedings of 22nd Annual Network Distributed System Security Symposium, 8-11 February 2015, San Diego, CA, USA, Copyright 2015 Internet Society.

server to phish its service subscribers. By poisoning the Link Discovery Service, an adversary can inject falsified links to create a black-hole route or launch a man-in-the-middle attack to eavesdrop/manipulate messages in the network. Our new attacks share some similarities in spirit to traditional spoofing attacks in legacy networks (e.g., ARP Poisoning Attack), however with significant differences in exploiting unique SDN vulnerabilities. According to our study, all major open source SDN controllers in the market (i.e., Floodlight, OpenIRIS, OpenDayLight, Beacon, Maestro, NOX, POX, and Ryu) are affected.

In order to mitigate such attacks, we investigate possible defense strategies. We note that it is difficult to simply use static configuration to solve the problem (similar to using static ARP entry for hosts or the port security feature for switches to address ARP poisoning attacks) because it requires tedious and error-prone manual effort and is not suitable for handling network dynamics, which is a valuable innovation of SDN. To better balance the security and usability, in this work, we propose TOPOGUARD, a new security extension to the existing OpenFlow controllers to provide automatic and real-time detection of network topology exploitation. By utilizing SDN-specific features, TOPOGUARD checks precondition and postcondition to verify the legitimacy of host migration and switch port property to prevent the Host Location Hijacking Attack and the Link Fabrication Attack.

To summarize, the contributions of this work include the following:

- We perform the first security analysis on the network visibility provided by SDN/OpenFlow Topology Management Service. In particular, we have discovered new vulnerabilities in the Device Tracking Service and Link Discovery Service in eight mainstream SDN/OpenFlow controllers.
- We propose Network Topology Poisoning Attacks to exploit the vulnerabilities we have found. Topology Poisoning Attacks can significantly infect the fundamental network visibility provided by SDN and cause serious security issues for upper services. We demonstrate the feasibility of those attacks both in the Mininet emulation environment and a hardware SDN testbed.

- We investigate the defense space and propose automatic mitigation approaches against Network Topology Poisoning Attacks, along with a prototype defense system, TOPOGUARD, to greatly enhance the security of the network visibility of SDN. Our evaluation shows that TOPOGUARD imposes only a negligible performance overhead.

2.2 Background and Security Analysis

In this section, we provide an introduction to basic operations in SDN/OpenFlow. Then, we present security analysis on network visibility provided in the existing SDN controllers. Finally, we introduce two novel attack vectors against network visibility as a motivation for TOPOGUARD to enhance the security of network visibility in SDN.

2.2.1 Background on Basic SDN Operations

Software-Defined Networking (SDN) is a new programmable network framework that decouples the control plane from the data plane. An SDN application in the control plane generates complex network functions such as computing a routing path, monitoring network behavior, and managing network access control. The data plane handles hardware-level network packet processing based on high-level policies from the control plane. SDN enables users to design and distribute innovative flow handling and network control algorithms conveniently and add much more intelligence and flexibility to the control plane. We can implement new control functions or protocols just as writing a normal application (analogous to writing an app for smartphone/Android OS). OpenFlow, as a leading reference implementation of SDN, defines the communication protocol between the control plane and the data plane. An OpenFlow switch must establish a TCP connection (with an option of TLS/SSL) to the OpenFlow controller before exchanging symmetric/asynchronous OpenFlow messages. When a new packet comes into an OpenFlow switch, the switch checks if the packet matches any existing flow rules. If so, the switch will process the packet based on the matching rule with the highest priority. Otherwise, the switch sends a *Packet-In* OpenFlow message to the OpenFlow controller to ask for proper actions according to network policies specified in the SDN apps. Once the specific decision is made, the OpenFlow controller either issues a

Packet-Out message for the one-time packet processing or instructs the OpenFlow switch to install new flow rules by sending a *Flow-Mod* message. In addition, whenever any change on a switch port is detected, a *Port-Status* OpenFlow message must be sent to the controller.

2.2.2 Security Analysis on Network Visibility in SDN

Different from legacy networks, topology management is unique in SDN networks due to the newly added, logically centralized network controller. In order to facilitate network management and programmability, the SDN controller maintains topology information and provides such network visibility to upper services/apps, as shown in Figure 2.1. More importantly, not only all controllers use the same topology discovery mechanism, but also both core controller components and SDN applications are tightly coupled with the topology information. The more OpenFlow applications are developed, the more critical dependencies would affect the whole components in the controller.

Generally, in an SDN/OpenFlow network, topology management includes three parts: (1) switch discovery, (2) host discovery, (3) internal link (i.e., switch-to-switch link) discovery. The switch discovery does not require any additional protocol since when an OpenFlow switch establishes a connection to the OpenFlow controller, the switch information should be stored in the OpenFlow controller for future management. When a switch receives any packet from a host, and it does not match any flow entry in the flow table, a *Packet-In* message encapsulating the packet is sent to the OpenFlow controller. The OpenFlow controller then learns the information about the host and its location (i.e., the corresponding attached switch port) from the message. For internal link discovery, the OpenFlow controller herein utilizes OpenFlow Discovery Protocol (OFDP). In this work, we mainly focus on the Host Tracking Service and Link Discovery Service inside the OpenFlow controller.

Host Tracking Service. Inside an OpenFlow controller, Host Profile is maintained to track the location of a host. There are significant advantages to Host Tracking. For example, in a data center, it is tedious and error-prone to manually maintain the locations of virtual machines due to their frequent migration. Also, as demonstrated in [30], the OpenFlow controller with host

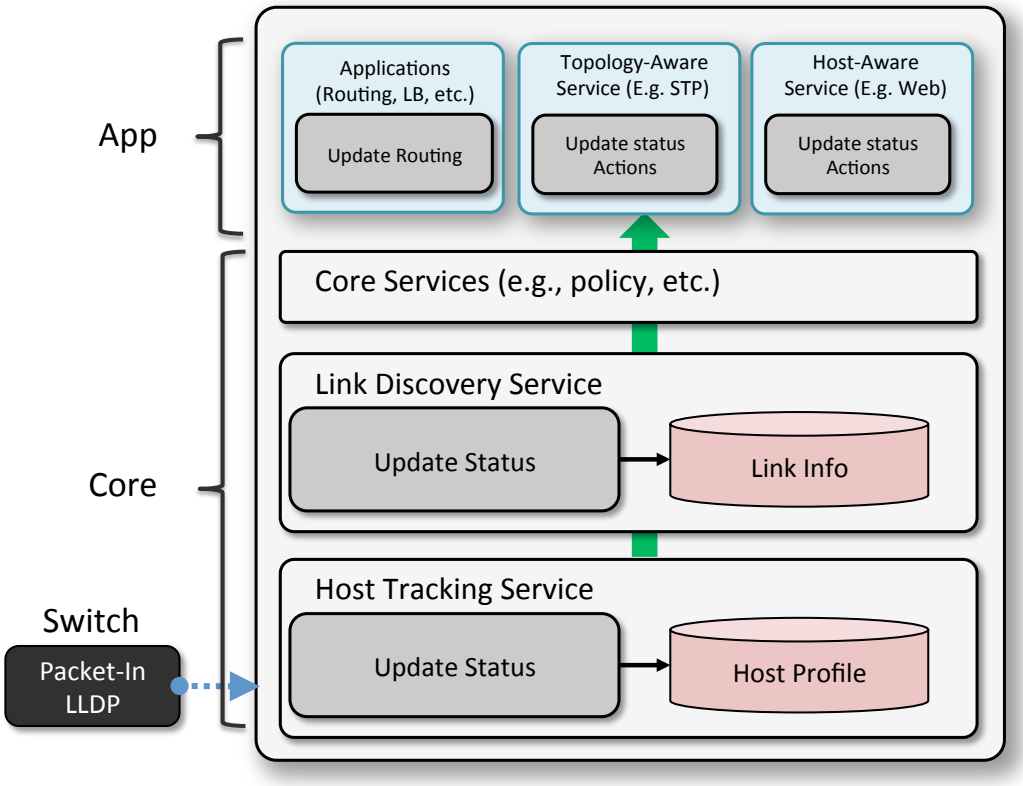


Figure 2.1: Service dependencies among layered controller components

location tracking can provide seamless handoff among different access points. In this regard, the Host Tracking Service (HTS) in the controller is to provide an easy way to guarantee flexible network dynamics by dynamically probing *Packet-In* messages and updating Host Profiles.

Let us take a close look at how an OpenFlow controller tracks the dynamics of host devices. In an OpenFlow network, the OpenFlow controller reactively listens to *Packet-In* messages to maintain Host Profile. During this procedure, the OpenFlow controller mainly handles two relevant host events (i.e., JOIN and MOVE). The scenario for the first event is that, when the OpenFlow controller fails to locate an existing Host Profile according to the information from incoming *Packet-In* messages, it creates a new Host Profile. In such a case, the controller assumes a new host joins the network. The second scenario occurs when the OpenFlow controller successfully locates a Host Profile but finds mismatched location information between the Host Profile and *Packet-In* messages. In the case, it assumes the host has moved to a new location and then updates the location

information inside the corresponding Host Profile.

Table 2.1 shows Host Tracking Services in current OpenFlow controller platforms. In order to handle host mobility, the existing OpenFLow controllers maintain a profile for each host. In detail, the Host Profile includes: (1) MAC address, (2) IP address, and (3) Location information (i.e., the DPID and the port number of the attached switch as well as the last seen timestamp). Normally, a Host Profile is indexed by the MAC address. Floodlight, for example, indexes the Host Profile with MAC address and VLAN ID. Beacon and the old version of Host Tracking Service in OpenDayLight support both MAC address and IP address as the index.

Vulnerabilities in Host Tracking Service. Host Tracking Service in the OpenFlow controllers maintains Host Profile for each end host to track network mobility. As long as hosts (or virtual machines) migrate, HTS can quickly react to such event. In particular, HTS recognizes the motion of hosts by monitoring *Packet-In* messages. Once being aware that a particular host migrates to a new location, i.e., DPID or ingress Port ID is different from the corresponding entry of the Host Profile, HTS updates Host Profile and optionally raises a HOST_MOVE event to its subscriber services. However, such an update mechanism is vulnerable due to the ignorance of authentication. In order to investigate security issues when HTS updates Host Profile, we manually analyze the source code of HTS in current mainstream OpenFlow controllers. Our study shows that existing OpenFlow controllers have few security restrictions on host location update. For instance, Floodlight and the old version of OpenDayLight controller provides an empty-shell API, called *isEntityAllowed*, which accepts every host location update rather than blocking possible spoofing attacks. The POX controller throws a warning if the observed time for device migration is less than a minimum expected time (60 seconds by default). However, we assume that such simple verification is easy to bypass if the adversary recognizes this feature in advance. The lack of consideration on security provides an opportunity for an adversary to tamper host location information by simply impersonating the target host. What is worse, all OpenFlow controllers have a routing module that utilizes the host location information to make the packet forwarding decision. That is, if an adversary can tamper the location information, he/she has the potential to hijack the traffic

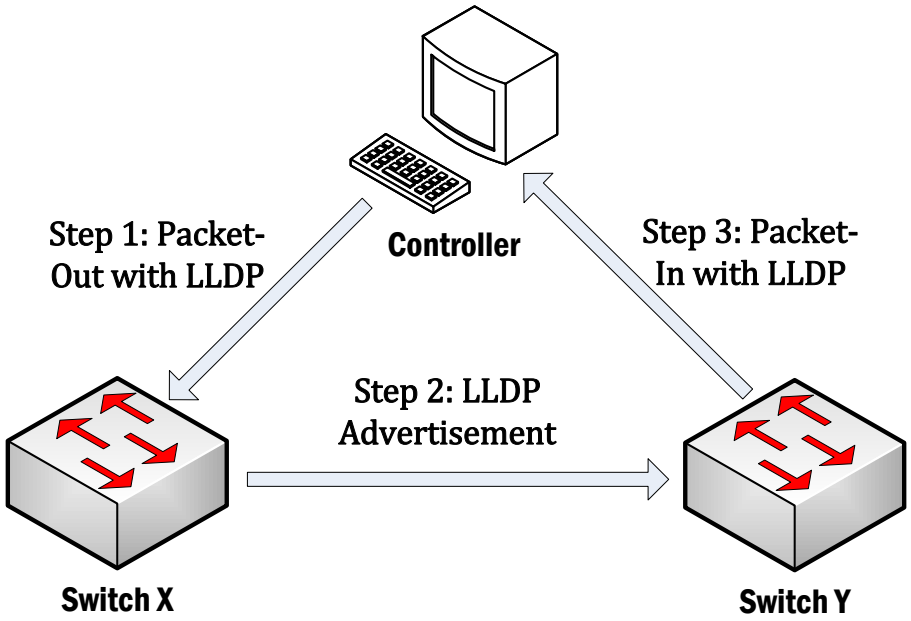


Figure 2.2: Link discovery procedure in OpenFlow networks

towards the host.

Table 2.1: Topology management services

Controller Platform	Link Discovery Service	TLVs	Host Tracking Service	Host Profile
Ryu	switches.py	DPID, Port ID, TTL	host_tracker.py	MAC, ***IP***, Location
Maestro	DiscoveryApp.java	DPID, Port ID, TTL	LocationManagementApp.java	***MAC***, Location
NOX	discovery.py	DPID, Port ID, TTL	hosttracker.cc	***MAC***, Location
POX	discovery.py	DPID, Port ID, TTL, System Description	host_tracker.py	***MAC***, IP, Location
Floodlight	LinkDiscoveryManager.java	DPID, Port ID, TTL, System Description	DeviceManagerImpl.java	***MAC***, ***VLAN ID***, IP, Location
OpenDayLight	DiscoveryService.java	DPID, Port ID, TTL, System Description	DeviceManagerImpl.java	***MAC***, ***VLAN ID***, IP, Location
OpenIRIS	OFMLinkDiscovery.java	DPID, Port ID, TTL, System Description	OFMDeviceManager.java	***MAC***, ***VLAN ID***, IP, Location
Beacon	TopologyImpl.java	DPID, Port ID, TTL, Full Version of DPID	DeviceManagerImpl.java	***MAC***, VLAN ID, ***IP***, Location

Link Discovery Service. To dynamically discover topology, the Link Discovery Service (LDS) inside SDN controllers uses Open Flow Discovery Protocol (OFDP), which refers to LLDP (Link Layer Discovery Protocol) packets to detect internal links between switches.

Figure 2.2 depicts the link discovery procedure in an OpenFlow network. Note that here we illustrate only a unidirectional link discovery for simplicity (the discovery of the opposite link is performed in a similar fashion). Initially, the OpenFlow controller sends out *Packet-Out* messages to Switch X with the payload of a controller-specific LLDP packet. The payload of the LLDP

packet contains DPID and the output port of Switch X. Upon receiving the LLDP packet, Switch X advertises it to all other ports in a broadcast manner. Typically, in an OpenFlow network, this kind of broadcast is achieved by iterative transmissions of one LLDP packet to each broadcast-enabled port of a switch. Then, the next-hop Switch Y, driven by its firmware or under explicit instructions of the attached OpenFlow controller, reports the incoming LLDP packet to the controller with the ingress Port ID and DPID of Switch Y via a *Packet-In* message. When receiving the *Packet-In* message from Switch Y, the OpenFlow controller can detect a unidirectional link from Switch X to Switch Y. Table 2.1 shows link discovery components in existing OpenFlow controller platforms. We find that all of these controllers embrace the internal link discovery procedure as we describe above.

In addition to the internal link discovery, some OpenFlow controller implementations, e.g., Floodlight and OpenIRIS, also propose a scheme to detect *multi-hop links*, which refers to links traverse across a Non-OpenFlow island. In order to detect such links, Floodlight leverages BDDP packets(i.e., a broadcast version of LLDP packets with a broadcast destination MAC address) to overcome unpredictable forwarding behaviors of Non-OpenFlow switches.

Vulnerabilities in Link Discovery Service. To build the entire network topology and handle dynamics of a network, OpenFlow adopts OFDP for topology management. Typically, OpenFlow controllers utilize LLDP packets to discover links among OpenFlow switches. However, there exist security flaws during the link discovery procedure.

The LDS in OpenFlow controllers is subject to two invariants: 1) The integrity/origin of LLDP packets must be ensured during the Link Discovery procedure; 2) The propagation path of LLDP packets can only contain OpenFlow-enabled switches. Unfortunately, those two security invariants are poorly enforced in the current instantiations of OpenFlow controllers. In our study, we find that the syntax of LLDP packets varies among different OpenFlow controller platforms. For example, POX and Floodlight use an integer variable to represent the port number of a remote switch, whereas the form of the port number in OpenDayLight is the ASCII value of characters. In addition, some OpenFlow controllers add extra TLVs (Type-Length-Values), e.g., system de-

scription, to enrich the semantics of LLDP packets. The controller-uniqueness of LLDP packets to some extent protects the LLDP "origin invariant." However, we argue that it is not enough when taking into account the open source nature of OpenFlow controllers and simple semantics of LLDP. Also, the Floodlight controller adds an origin authenticator as an extra TLV of LLDP packets to verify the origin of LLDP packets. However, the authenticator keeps unchanged after the setup of Floodlight controllers, which allows an adversary to violate the origin property. More seriously, we find that there are no mechanisms in current OpenFlow controllers to ensure the integrity of LLDP packets.

In our study, we also find some OpenFlow controllers, e.g., Floodlight and OpenDaylight, provide an API *suppressLinkDiscovery* to block LLDP propagation to particular ports connected to hosts. This kind of method is similar to the BPDU Guard security feature in legacy Ethernet switches, which prevents BPDU frames from sending to those ports enabled with the PortFast feature (i.e., manual configuration of switch ports connected to hosts). However, depending only on static port settings is not enough for diverse OpenFlow network environments, varying from a home network to an enterprise or cloud/data-center network and from stationary networks to mobile networks.

In order to deceive the LDS, an adversary can violate the "integrity/origin invariant" and "path invariant" of LLDP packets. In particular, the adversary originates falsified LLDP packets or simply relays LLDP packets between two switches to fabricate a non-existing internal link. At first glance, it does not seem practical to inject arbitrary packets into the network from hosts or virtual machines because they are usually isolated by specific mechanisms, e.g., VLAN and Firewall. However, it appears feasible for hosts and virtual machines to inject or relay LLDP packets in OpenFlow networks. The OpenFlow networks allow LLDP packets to be sent outside all switch ports to track internal links between switches dynamically. Thus, the current design of OpenFlow controllers accepts LLDP packets from each switch port, even though it is connected to a host, which leaves a room for an adversary to inject fake internal links on compromised hosts or virtual machines.

2.2.3 Novel Attacks against the Network Visibility in SDN

In this part, we present two novel attack vectors against network visibility in SDN by exploiting vulnerabilities in Topology Management Services.

Threat Model. We assume an adversary possesses one or more compromised hosts or virtual machines (e.g., through malware infection) in the SDN/OpenFlow network and has the read and write privilege on packets in the operating system.[2] Note that, in this work, we assume the transmission of OpenFlow messages via the control channel can be properly protected by SSL/TLS.

Experimental Environment. Furthermore, we demonstrate the SDN-specific Network Topology Poisoning Attacks both in Mininet 2.0 [31] and a physical environment (with hardware OpenFlow switches). Mininet 2.0 is widely used for emulating an OpenFlow network environment. Our hardware testbed includes several OpenFlow-enabled hardware: TP-LINK(TL-WR1043ND) and LINKSYS(WRT54G) which run OpenWRT firmware with an OpenFlow extension and PCs with Intel Core2 Quad processor and 2GB memory.

Host Location Hijacking Attack. Here, we propose an attacking strategy where the adversary crafts packets with the same identifier of the target host. Once receiving the spoofed packet, the OpenFlow controller will be tricked to believe that the target host moves to a new location, which actually is the attacker's location. As a result, future traffic to the target is hijacked by the adversary. Next, we introduce a practical example of harvesting web clients by exploiting the vulnerability in HTS, as shown in Figure 2.3.

In order to conduct a Web Clients Harvesting Attack, we first need to retrieve the identifier of the target. From Table 2.1, we find that the host identifier varies among MAC address, VLAN ID, and IP address depending on the platform and version of OpenFlow controllers. It is trivial to know the IP address if we have already chosen an attacking target. Besides, the VLAN ID is normally unused during the update procedure of Host Profile. On the other hand, as MAC address is regarded as the (or part of) identifier for hosts in most OpenFlow controllers (except for Ryu), we can use ARP request packets to probe the MAC address of our target. Note that such a simple

[2] In the extreme case, the adversary can be an insider.

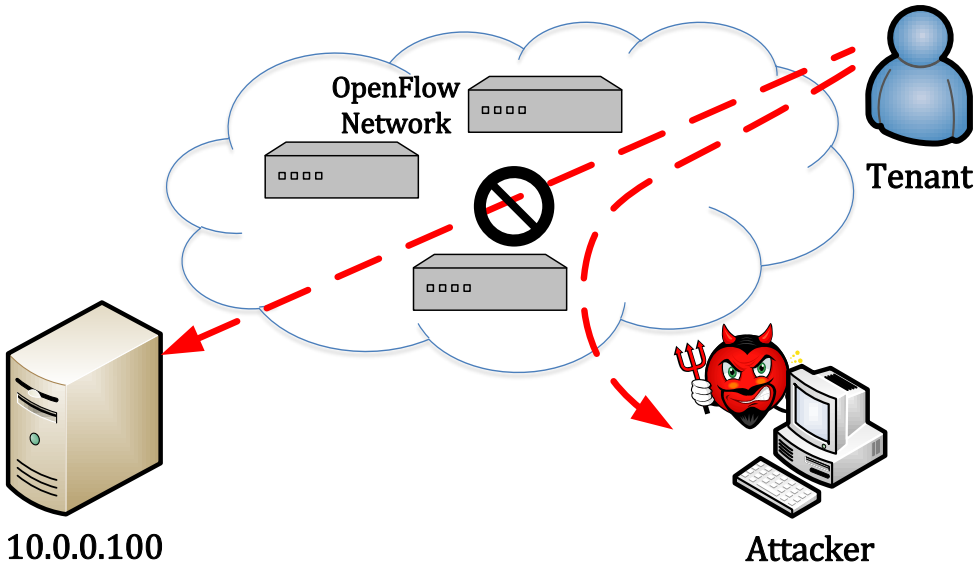


Figure 2.3: Attacker impersonates a specific web server to phish users

probe method is feasible because the OpenFlow network does not change the source MAC address during packet transmission.

In addition, one difficulty to successfully exploit HTS lies in that the adversary needs to race with the target host because any traffic initiated from the target host can correct host location information in the controller. To overcome the non-determinism of such situation, we could set our target as a server. This is because a server normally runs in a passive mode, i.e., it opens a specific port(s) and waits for remote connections from clients.

In this thesis, we launched a Host Location Hijacking Attack in our experimental environment. We chose Floodlight (master) as the OpenFlow controller, atop which the Host Tracking Service and Shortest Path Routing Service are enabled by default. We deployed an Apache2 [32] web server with IP address "10.0.0.100" and several hosts in our customized OpenFlow topology. The reachability test is shown in Figure 2.4(a), that is, before we launch the Host Location Hijacking Attack, clients can visit the genuine web server with our assigned IP address. Upon a compromised host, we also run a Web service and send ARP request to probe the corresponding MAC address of "10.0.0.100". We then use Scapy [33] to periodically inject fake packets in the name of our target

(a) Connected to Genuine Server

(b) Hijacked by Malicious Server

Figure 2.4: Web impersonation attack

(the genuine web server “10.0.0.100”). After that, we find the new coming client attempting to visit the web server “10.0.0.100" is directed to the malicious server, as shown in Figure 2.4(b).

Link Fabrication Attack. In this part, we show how an adversary can fabricate a link into the network topology to threaten normal network activities in two ways, i.e., Fake LLDP Injection and LLDP Relay.

Firstly, an adversary can generate fake LLDP packets into an OpenFlow network to announce bogus internal links between two switches. By monitoring the traffic from OpenFlow switches, the adversary can obtain the genuine LLDP packet. Afterward, he/she can violate the origin invariant of an LLDP packet, while OpenFlow controllers leverage specific syntax and extra TLVs for verification. Due to the open source nature of most OpenFlow controllers, the adversary, can find out the reference LLDP syntax. Although the source code of OpenFlow controllers could be veiled and a network administrator could customize the source code, it is also available to decode the LLDP packets by using differential tools. Moreover, as described above, the weak authenticator of LLDP packets imposed by several OpenFlow controllers can be bypassed. As long as the adversary acquires the genuine LLDP packet along with its syntax, he/she can modify the specific contents of the LLDP packet, e.g., the DPID field or the port number field, and launch the Link Fabrication Attack. In order to circumvent the possible anomalous traffic detection, the adversary could tune the LLDP injecting rate to the LLDP sending rate monitored from the OpenFlow controller.

Instead of injecting forged LLDP packets, a stronger adversary can also fabricate internal links in a relay fashion (without packet modification). That is, when receiving an LLDP packet from

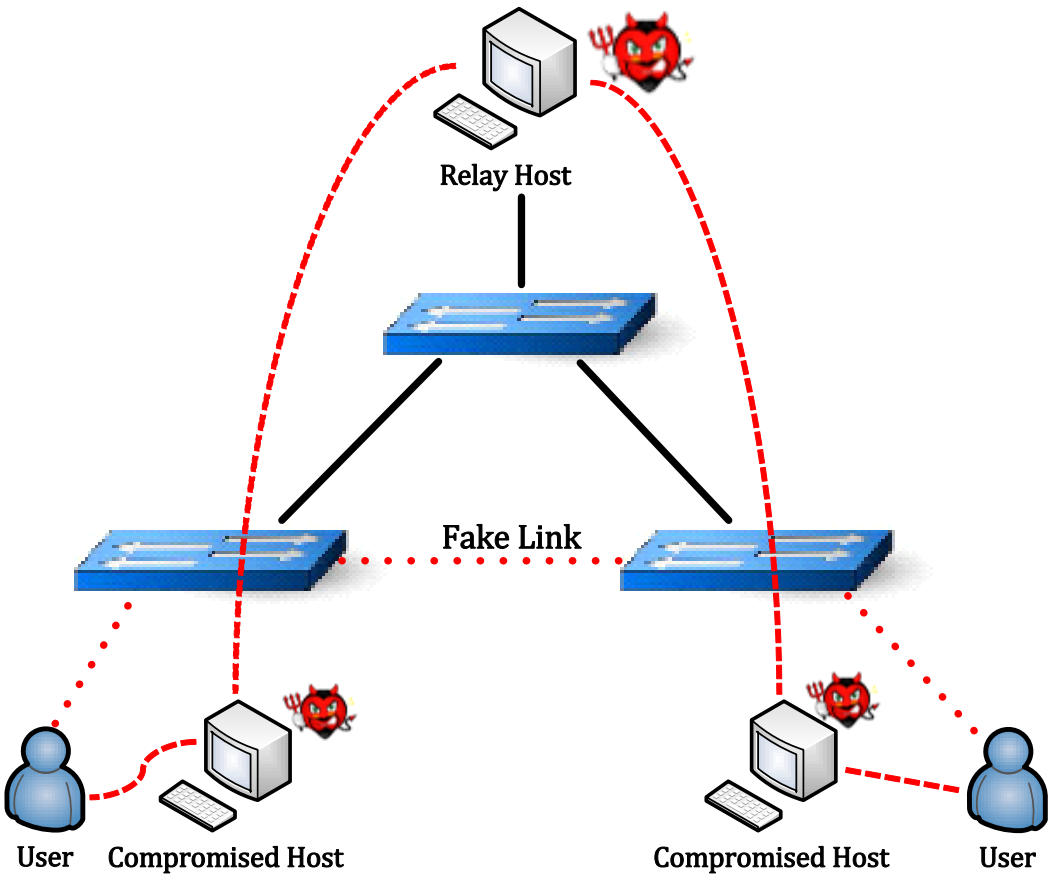


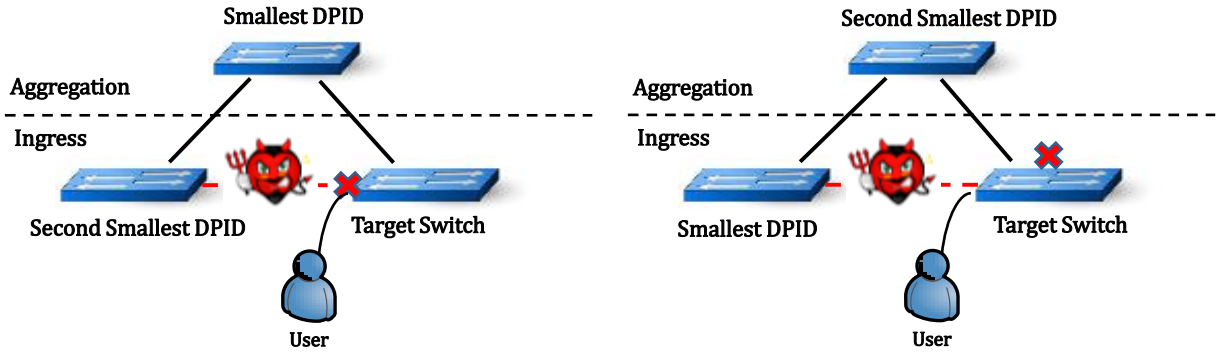
Figure 2.5: Link fabrication in an LLDP relay manner

one target switch, the adversary repeats it to another target switch without any modification. In the case, the adversary constructs a fake topology view to the OpenFlow controller as if there is an internal link between those two target switches. This kind of fake link injection incurs future attack possibilities which we will describe more in detail as follows.

Here, we discuss two ways to build a communication channel to relay LLDP packets, i.e., by physical links and by a tunnel. An intuitive relay method is that an adversary sets up physical links (e.g., cable or wireless) between two switches. If this is not feasible, the adversary can use another more feasible approach, which relays LLDP packets by reusing the existing OpenFlow network infrastructure as illustrated in Figure 2.5. Particularly, the dotted line is the communication channel between two users in the view of an OpenFlow controller, whereas the dashed line is the actual traffic route. To successfully launch an LLDP relay attack, the adversary first needs to

find a suitable relay host(s), which can be achieved by a connection test. Another thing we need to note is that some OpenFlow controllers, e.g., Floodlight and POX, disable the Host Tracking Service on internal link switch ports, which hinders the deployment of the LLDP relay channel. However, we cannot ignore the tunnel-based LLDP relay attack on those controllers in a hybrid network scenario (i.e., a network contains both OpenFlow islands and Non-OpenFlow islands), as the OpenFlow controller can hardly stop Host Tracking Service on the Multi-hop link ports (i.e., switch port outgoing to another Non-OpenFlow switch).

Next, we illustrate two attack possibilities on the top of Link Fabrication Attack, i.e., Denial of Service attack and man-in-the-middle attack.



(a) The chosen switch has the second smallest DPID (b) The chosen switch has the smallest DPID

Figure 2.6: Denial of service attack

Denial of Service Attack. To prevent a broadcast storm and save energy, OpenFlow controllers provide Spanning Tree Service. When any topology update occurs, Spanning Tree Service is triggered to block those redundant ports. However, this capability can be leveraged by an adversary to launch a Denial of Service attack. In particular, by injecting a fake link into existing topology, the adversary can borrow the knife of Spanning Tree Service to "kill" normal switch ports.

One challenge to launch this type of attack is the non-deterministic characteristics of Spanning Tree Calculation after fake link injection. We note that the Spanning Tree Algorithm always ex-

cludes the link that connects the largest DPID switches. Hence, we introduce an attack strategy tailored to a practical scenario, where an adversary possesses several compromised hosts connected to ingress switches. By listening to LLDP packets, the adversary can acquire the DPIDs of two ingress switches. Then, the adversary controls the compromised host which connects to the ingress switch with a lower DPID and injects a fake LLDP to announce a link with the target switch. As a result, there may be two consequences: if the DPID of the aggregation switch is smaller than that of our chosen switch, the adversary could shut down an arbitrary port of the target switch, as shown in Figure 2.6(a); otherwise, if the chosen switch has the smallest DPID, the link between the target switch and the aggregation switch is excluded from the spanning tree and also the corresponding ports are blocked, as shown in Figure 2.6(b).

We demonstrated a Denial of Service attack in our experimental environment. We chose POX as the OpenFlow controller, enabling routing module (l2_learning.py), link discovery module (discovery.py) and spanning tree module (spanning_tree.py). Note that the action for non-spanning-tree ports was configured as Port_Down. Then, we deployed a FatTree-like topology, where we controlled two hosts connecting to two sibling ingress switches. We ran Wireshark with the OpenFlow Dissector extension [34], which helps to parse OpenFlow messages, and dumped the *Packet-Out* messages with the payload of LLDP packets. We also ran an attacking script to craft fake LLDP packets based on the dumped genuine LLDP packets and injected them to the switch with the smaller DPID. As a result, we noticed that the users who are connected to our target switch port could not access the network resource anymore.

2.3 Countermeasure Analysis

2.3.1 Static Defense Strategies

To defeat the proposed Network Topology Poisoning Attacks in SDN networks, we can have two major types of defense strategies: static or dynamic. The static solution is to manually configure/manage the host location and link information beforehand (e.g., assign a host identifier such as a MAC address to a specific switch port), and then manually verify and modify whenever there are

changes (new addition or removal). However, this defense is obviously not attractive as the manual management is tedious, error-prone and not scalable in practice. In particular, it is not suitable for SDN networks, in which dynamics could be common and the scalability is important. Thus, in the following sections, we mainly focus on discussing dynamic defense strategies, as briefly summarized in Table 2.3. We will further introduce our proposed new defense system, TOPOGUARD, and evaluate its effectiveness and performance in the later section.

Table 2.2: Defense capabilities

	Host Migration	**Comparative Feasibility**	**Integirty/Origin Invariant of LLDP**	**Path Invariant of LLDP**
Authentication	Yes	Low	Yes	No
Verification	Yes	High	No	Yes

2.3.2 Dynamic Defense Strategies against Host Location Hijack

The problem of Host Location Hijacking lies in that OpenFlow controllers fail to verify the host identifier when the location of a host is updated. To tackle this issue, we discuss two possible mitigation methods which can secure HTS in OpenFlow controllers as well as dynamically track network mobility.

Authenticate Host Entity. A cryptographic solution to this problem is to authenticate a host by adding additional public-key infrastructure. In particular, when a host needs to change its location, it encodes the new location information into an unused field of packet (e.g., VLAN ID or ToS) with the encryption using its private key. This solution seems decent to prevent malicious host profile falsification because it is not practical for an adversary to acquire the private key of the target host. However, there are several restrictions that make it hard to be feasible in practice. First, it begets additional storage overhead for keeping public keys in the OpenFlow controller side as well as computation overhead for handling each *Packet-In* message. The management of all keys of hosts and the dynamic addition/removal also bring a lot of overhead and cost. Moreover, this method

requires to modify the implementation of every host, which is tedious and difficult in practical deployment.

Verify the Legitimacy of Host Migration. Another lightweight solution we propose is to verify the conditions of host migration. The idea is based on our two observations. First, the precondition of a host migration is that the OpenFlow controller must receive a Port_Down signal before the host migration finishes. Second, the postcondition of a host migration is that the host entity is unreachable in the previous location after the completion of the host migration. Consequently, based on this cause-and-effect relation, we can verify the legitimacy of the host migration by checking the precondition and postcondition. This method also adds performance overhead for *Packet-In* message processing, but compared to Host Entity Authentication, it is lighter and more feasible. In this work, we adopt this verification approach to secure the host migration.

2.3.3 Dynamic Defense Strategies against Link Fabrication

As mentioned earlier, the root causes of the Link Fabrication attack can be summarized as: 1) The integrity/origin of an LLDP packet can be violated during the link discovery procedure in OpenFlow networks; 2) The compromised hosts can involve in the LLDP propagation path. To fix those security omissions, we propose two approaches that can secure the Link Discovery procedure without the burden of manual effort.

Authentication for LLDP packets. The first security omission exploited by an adversary is that the OpenFlow controller fails to verify the integrity of LLDP packets. Also, the adversary can defeat the verification of the origin in current OpenFlow controllers as long as he/she is able to receive LLDP packets from the connected switch. One solution to this problem is to add extra authenticator TLVs in the LLDP packet. Especially, we can add a controller-signed TLV into the LLDP packet and check the signature when receiving the LLDP packets. The signature TLV is calculated over the semantics of the LLDP packet (i.e., DPID and Port number). In this case, the adversary can hardly manipulate the LLDP packets. However, this approach suffers from the fact that it fails to defend against the Link Fabrication attack in an LLDP relay/tunneling manner.

Verification for Switch Port Property. Another security invariant of the OpenFlow link dis-

covery procedure is that no hosts can participate in the LLDP propagation. An idea to mitigate the relay-based Link Fabrication is to check if any host resides inside the LLDP propagation, e.g., we can add some extra logic to track the traffic coming from each switch port to decide which device is connected to the port. If OpenFlow controllers detect host-generated traffic (e.g., DNS) from a specific switch port, we set the *Device Type* of that port as HOST. Otherwise, we assign those switch ports as SWITCH when LLDP packets are received from those ports. In OpenFlow networks, those two categories are mutually exclusive because LLDP can only transmit on switch internal link ports and ports connected to the OpenFlow controller [3]. One assumption of this method lies in that the compromised machine is not an actual switch thus will generate regular host-generated traffic (e.g., ARP, DNS). This assumption is reasonable and it holds in most cases in practice. While a powerful adversary could theoretically disable all host-generated traffic in compromised hosts or virtual machines, it could also make the machine somewhat non-functional, at least for some normal networking activities/operations, and such non-functional anomaly could be easily noticed by the normal machine user, thus expose the existence of the adversary.

Finally, we note that in the case the adversary mutes all host-generated traffic, our aforementioned switch port property verification may not work. From the controller perspective, the attacking host can act just as a part of a cable, which is very difficult to discover by layer 2 or layer 3 security mechanisms. We could resort to verify some physical layer characteristics (e.g., [35]) to differentiate whether the attached device hardware is a switch or a machine, which is out of the scope of this work.

2.4 System Design

In this section, we detail the design and implementation of a new security extension for the OpenFlow controller, called TOPOGUARD, to protect the SDN network visibility from Network Topology Poisoning Attacks. TOPOGUARD is certainly not perfect. Our goal is to provide an automatic tool that (i) has a good balance between usability and security, and (ii) can be easily

[3] In this work, we consider the control channel of OpenFlow networks could be properly under the protection of SSL/TLS.

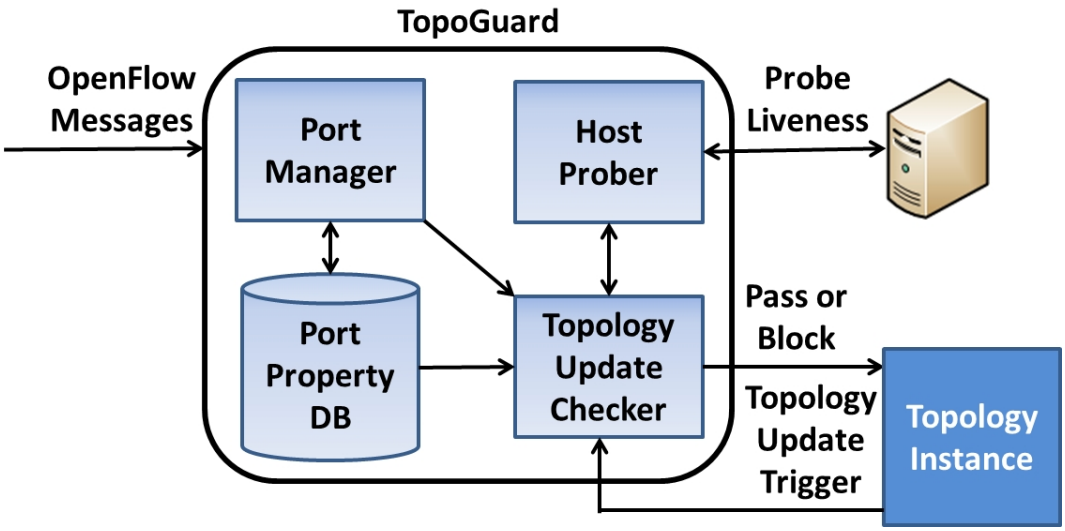


Figure 2.7: The architecture of TOPOGUARD

integrated into current mainstream OpenFlow controllers for immediate protection.

2.4.1 System Architecture

The basic idea of TOPOGUARD is to secure OpenFlow controllers by fixing security omissions as described in the previous section. In TOPOGUARD, we design Topology Update Checker to automatically validate the update of network topology, which is dependent on the information provided by Port Manager and Host Tracker.

Figure 2.7 illustrates the architecture of our defense system. The Topology Update Checker verifies the legitimacy of a host migration, the integrity/origin of an LLDP packet and switch port property once detecting a topology update. Specifically, the Port Manager surveils OpenFlow messages to track dynamics of switch ports, which are stored in the Port Property. Afterwards, the Port Property is used to reason about the trustworthiness of a topology update. The Host Prober module is to test the liveness of the host in the specific location of the OpenFlow network, which also provides forensics to judge the host migration.

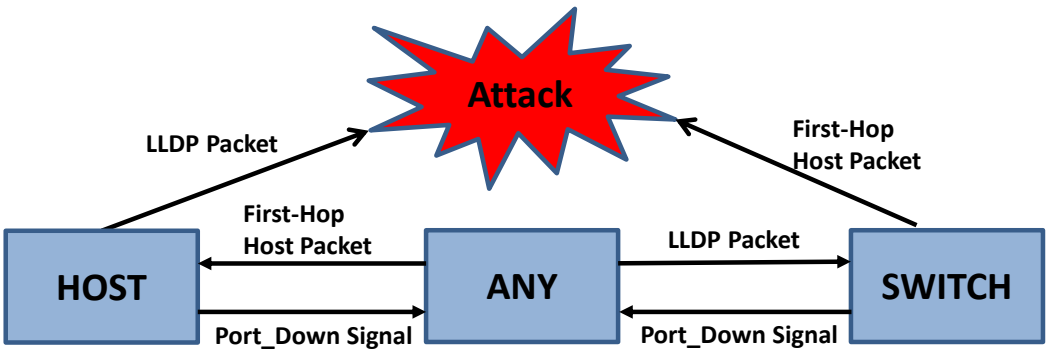


Figure 2.8: The transition graph of device type

2.4.2 System Components

Port Property Management. In order to reason about the validness of a topology update, we profile extra properties for each switch port in an OpenFlow controller. These properties include: *Device Type*, *Host List* and *SHUT_DOWN_FLAG*. The *Device Type* depicts which type of device a particular switch port connects to. The value could be ANY, SWITCH and HOST. As illustrated in Figure 2.8, The initial value of *Device Type* is ANY, which will turn to SWITCH or HOST based on following traffic. When Port Manager receives LLDP packets from a switch port with *Device Type* of ANY, it changes its type to SWITCH. Similarly, the *Device Type* of the switch port is set to HOST when Port Manager receives any first-hop host traffic, i.e., the host identity of the traffic is detected by the OpenFlow controller for the first time. In contrast, the HOST and SWITCH port are set back to ANY when receiving a Port_Down signal indicated in *Port-Status* messages. If Port Manager detects an LLDP packet from a HOST port or a first-hop host packet from a SWITCH port, it raises an attack alert and notifies Topology Update Checker to prevent the relevant topology update. The intuition behind this defense approach is that an LLDP packet is only designed to traverse through switch internal link ports in the data plane.

One challenge in port property management is how to decide the *Device Type* of a port as HOST. An intuitive solution in the OpenFlow networks is to monitor *Packet-In* messages from the switch port to detect host-generated traffic (e.g., ARP and DNS). After detecting host-generated

traffic, we consider the port is connected to a host and change its *Device Type* as HOST. However, in our study, we find that different OpenFlow switches may issue multiple replicas of *Packet-In* messages for a specific host flow, i.e., the OpenFlow controller would receive host traffic from switch internal link ports. It could be due to the race condition scenario or specialized packet processing logic of OpenFlow controller applications. To solve this problem, we keep tracking the first-hop host traffic. Especially, we maintain *Host List* in the Port Property for each switch port, which contains host entities (in the form of a MAC address). When receiving *Packet-In* messages, the Port Manager locates the host entity in the existing *Host List* of Port Property. if the host entity is not found, the traffic is regarded as first-hop traffic and the source MAC address is recorded in the *Host List* of Port Property of the ingress switch port. Also, we need to handle network dynamics such as the *Set-Field* action in the OpenFlow flow rule, because any modification of the source MAC address during packet transmission can cause misclassification of first-hop traffic. For this, we also maintain a host-MAC alias map when we observe some flow rule modifying the source MAC address.

Another purpose of keeping *Host List* is to verify the trustworthiness of a host migration. The precondition for a host migration is that the OpenFlow controller receives a Port_Down signal before the host migration finishes. At this point, we set *SHUT_DOWN_FLAG* for hosts in the *Host List* of a switch port once detecting the port is down. The *SHUT_DOWN_FLAG* can be disabled when Port Manager receives correlated host traffic from the port. Furthermore, we can validate the *SHUT_DOWN_FLAG* inside *Host List* for the verification of the host migration.

Host Prober. As the counterpart to checking the precondition of a host migration, we can also leverage Host Prober to verify the postcondition, i.e., the host is unreachable in the previous location after the host migration completes. To achieve this, the Host Prober issues a host probing packet, e.g., ICMP Echo Request, to the former location of the host and waits for a response within a reasonable timeout. The Host Prober sends out a *Packet-Out* message with the payload of a crafted ICMP packet and outputs it to a specific switch port. In order to ensure the successful delivery of the response, the Host Prober also installs a flow rule to direct the ICMP response back

to the OpenFlow controller. Also, to lower the overhead, in the current implementation, we set the response timeout as 1 second.

Topology Update Verification. The Topology Update Checker verifies the correctness of a topology update including a host migration and a new link discovery. When a host migration is detected, the checker references Port Property to check the precondition and instructs Host Prober to validate the postcondition. We note that the time overhead of checking the postcondition would be much higher than that of checking the precondition. In order to reduce the overall overhead, we can adopt a roll-back technique in Host Migration verification. That is, the Topology Update Checker updates a host location if the precondition is passed (*SHUT_DOWN_FLAG* for that host is enabled in Port Property of the former location) without waiting for the result of Host Prober. However, if the response of Host Prober indicates a malicious host migration, the Topology Update Checker withdraws the previous update and raises an attack alert. In this case, the time overhead for verifying the host migration only counts on validating the precondition.

The Topology Update Checker also verifies the link discovery. The first task is to ensure the LLDP integrity/origin. For this sake, we place a signature TLV into an LLDP packet, which is a cryptographic hash value of a DPID and Port number. As soon as a new link is discovered, the Topology Update Checker conducts extra verification logic for the signed hash TLV. Then, the Topology Update Checker detects if the host lies on the path of the LLDP propagation. This task is achieved by checking the *Device Type* of switch ports of the new link. As a result, any internal link update involved in the HOST port will be denied and trigger an attack alert.

2.5 Evaluation

We evaluated a prototype implementation of TOPOGUARD to examine its effectiveness and performance.

2.5.1 Prototype Implementation

We have developed a prototype implementation of TOPOGUARD on the master version of Floodlight. The Topology Update Checker, including Port Manager and Host Prober, works as

a Floodlight service and is approximately 700 lines of Java code. The Topology Update Checker implements *IDeviceListener* and *ILinkDiscoveryListener* to monitor an update event of the topology instance inside Floodlight controller, while the Port Manager implements *IOFSwitchListener* and *IOFMessageListener* to initiate and maintain Port Property for each switch port.

To ensure the origin and integrity of an LLDP packet, we also use a keyed-hash message authentication code (HMAC) as an optional TLV for LLDP packets. In particular, we utilize javax.crypto package and select SHA-256 as the hash function along with controller's secret key. This adds about 130 lines of code in Java.

2.5.2 Effectiveness

We first measured the effectiveness of our implementation against the Network Topology Poisoning Attacks discussed in Section 2.2.3. Our experiment is conducted in the OpenFlow network environment including the Floodlight controller with TOPOGUARD. We launched aforementioned Network Topology Poisoning attacks in the environment and testified the reactions of the fortified Floodlight controller by observing the console output.

Detecting Host Location Hijacking. An adversary can spoof the identity of a target host to hijack its location information inside OpenFlow controllers. Note that we assume the target host is not compromised by the adversary. With TOPOGUARD, the falsified host migration can be detected due to dissatisfaction of the precondition and the postcondition. That is, the Floodlight controller fails to receive a Port_Down message before receiving a host move event as shown in the first line in the red pane of Figure 2.9, and it succeeds in probing the target host in the previous location after receiving the host move event, as shown in the second line in the red pane of Figure 2.9.

Preventing Link Fabrication. An adversary can also falsify LLDP packets to fabricate non-existing links between switches. Under the radar of TOPOGUARD, the attempts to exploit the poor origin check and the omitted integrity assurance of the LLDP packets can be efficiently prevented. To ensure network dynamics, we do not manually block the LLDP packets on switch ports, i.e., the adversary is allowed to receive LLDP packets. However, once either a DPID or Port ID is manipulated by the adversary, the fortified LLDP handler can detect it and fire an alert. Note that

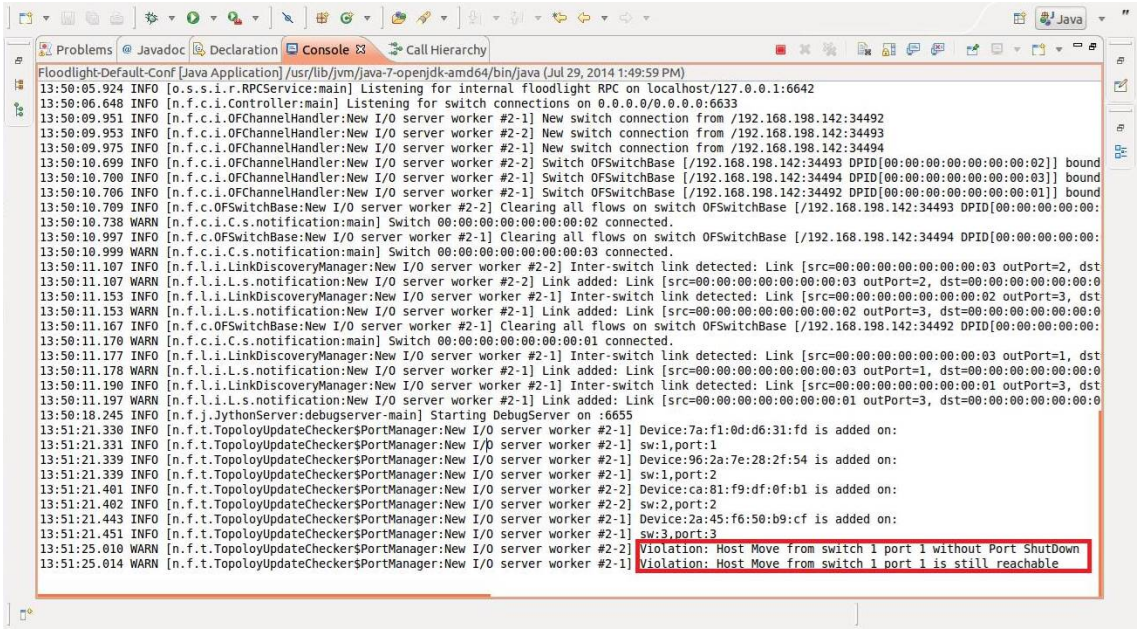


Figure 2.9: The detection of Host Location Hijacking attack

Table 2.3: HMAC overhead on the Floodlight controller

Link Discovery Snippet	**Impact of** TOPOGUARD **(Percentage)**	**Controller Overall Cost**
LLDP Construction(First time with computing HMAC)	0.431ms(80.4%)	0.536ms
LLDP Construction	0.005ms(2.92%)	0.171ms
LLDP Verification	0.005ms(1.64%)	0.304ms

we disable the port property verification while checking the integrity of LLDP packets because this LLDP falsification is also launched inside the data plane.

For another way of link fabrication, the adversary utilizes compromised hosts to relay LLDP packets between two target switches. When the compromised hosts start relaying LLDP packets, TOPOGUARD detects the violation of *Device Type* of particular ports, as shown in the red pane of Figure 2.10.

2.5.3 Performance Overhead

We further evaluated the performance of TOPOGUARD on Floodlight about the overhead over normal packet processing. In this experiment, we leverage Java System.nanoTime API to measure

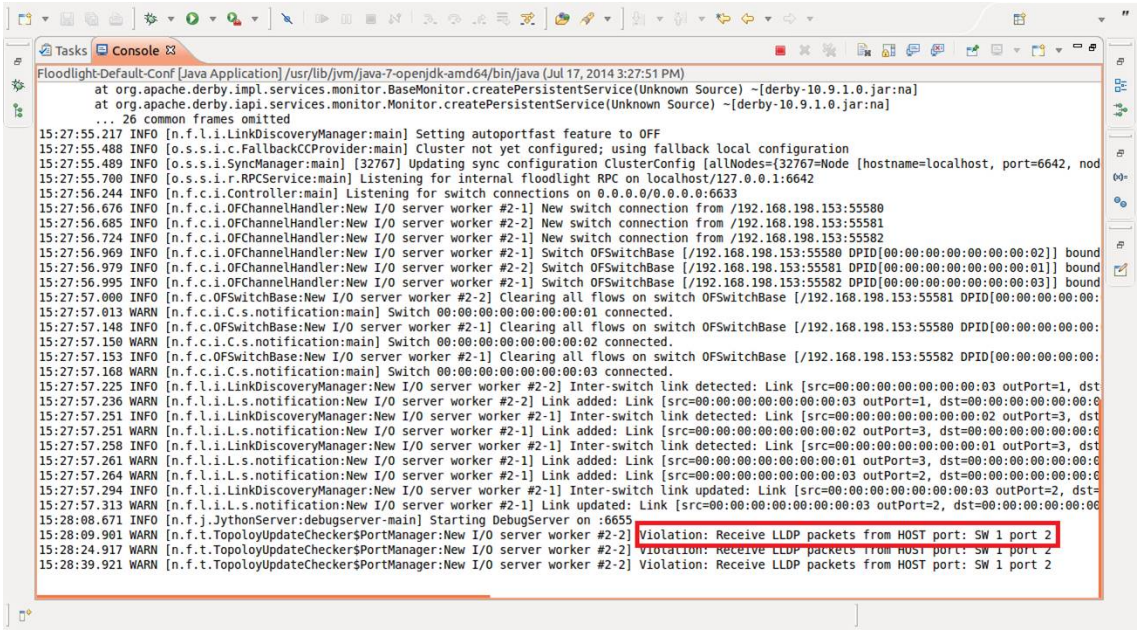


Figure 2.10: The detection of Link Fabrication Attack

the running time of program snippets, which provides a precision of 1 nanosecond. Note that the measurement is conducted after all modules of the Floodlight controller are completely booted.

The performance penalty imposed by TOPOGUARD mainly comes from the Link Discovery Module and the *Packet-In* message processing. Table 2.3 shows the average delay for TOPOGUARD added to the Floodlight controller on link discovery snippets, i.e., different functional blocks of Link Discovery Module. For the first round of the LLDP packets construction, the average overhead of TOPOGUARD is 0.431 ms, which accounts for 80.4% of overall LLDP construction time. However, we note that the following cost of TOPOGUARD imposed on the LLDP construction is much lower, which is about 0.005 ms and only accounts for 2.92% of overall LLDP construction time. The significant discrepancies stem from our implementation strategy because TOPOGUARD computes the HMAC value once and cache the computation result for the future construction and verification of LLDP packets. The strategy also lowers the impact of TOPOGUARD on the verification phase of LLDP packets, which is only about 0.005 ms. On the other hand, the Port Manager incurs a delay over the normal packets processing because it sits in the earlier stage in the OpenFlow message processing pipeline of the Floodlight controller than the Shortest Path

Routing Service and the Link Discovery Service. Accordingly, we also measure the time that the Port Manager spends on handling LLDP packets and host-generated packets. The result shows the average delay is 0.02 ms for the LLDP packets processing in the Link Discovery Service and 0.032 ms for the normal packets processing in the Shortest Path Routing Service. From the above result, we conclude that the impact of TOPOGUARD is negligible on the normal operation of Link Discovery Service of the Floodlight controller.

2.6 Related Work

In this section, we investigate security research in the SDN domain and related network visibility poisoning attacks in legacy networks.

Security Research for SDN networks. Several verification approaches are often used to debug and check network invariants. VeriFlow [36] presents a layer between the control plane and the data plane that monitors network state updates and verifies the violations of invariants dynamically at real time. NetPlumber [37] introduces a realtime network-wide policy checking tool using Header Space Analysis (HSA). NICE [38] uses model checking and symbolic execution to find network software bugs in OpenFlow applications. SOFT [39] introduces an approach for testing the interoperability of OpenFlow switches with reference implementations. [40] designs and presents the first machine-verified SDN controller based on NetCore [41]. A previous work introduced a verification tool that took the software program of a data plane as input and checked target properties [42]. These verification solutions only verify the logic correctness of the control plane and data plane, however fail to locate the network topology exploitations discussed in this work. One insight behind Network Topology Poisoning Attacks stems from the centralized network visibility that OpenFlow Controller offers to lessen onerous network management tasks. Unfortunately, our study in this work shows that this function could be exploited if not carefully designed, thereby incurring serious security threats.

To date, the security issues of SDN have been being widely discussed in both academia and industry. FortNox [43] introduces a SDN tunneling attack and presents two mechanisms, role-based authorization and security constraint enforcement, to solve corresponding security challenges. OF-

Table 2.4: Comparison between Host Location Hijacking and ARP Cache Poisoning

Attack Requirement	**OpenFlow Host Location Hijacking**	**ARP Cache Poisoning**
Attacker Location Restriction	Anywhere within the target's OpenFlow domain	Stay within the same broadcast domain with the target
Target Visibility	MAC Address and IP Address	Only IP Address
Attack Avenue	**OpenFlow Host Location Hijacking**	**ARP Cache Poisoning**
Falsified Packet Type	Almost every kind of packets	Only ARP packet
Attack Result	**OpenFlow Host Location Hijacking**	**ARP Cache Poisoning**
Hijack the Target Location	Yes	Yes

Table 2.5: Comparison between Link Fabrication and previous counterparts

Attack Requirement	**OFDP Link Fabrication**	**STP Mangling**	**OSPF Link Fabrication**	**OLSR Wormhole**
Compromising Routers/Switches	No	No	Yes	No
Defeating Neighbor Authentication	No	No	Yes	No
Attack Avenue	**OFDP Link Fabrication**	**STP Mangling**	**OSPF Link Fabrication**	**OLSR Wormhole**
Falsify Control Message	Yes	Yes	Yes	No
Relay Control Message	Yes	No	Yes	Yes
Attack Result	**OFDP Link Fabrication**	**STP Mangling**	**OSPF Link Fabrication**	**OLSR Wormhole**
Injecting False Link into Topology	Yes	No	Yes	Yes
Affected Service	All Topology-Based Services	STP	Routing	Routing

RHM [44] proposes OpenFlow Random Host Mutation to dynamically mutates IP addresses of hosts inside an LAN network. FRESCO [11] introduces an OpenFlow security application development framework which provides modular composable security services for application developers. SDN Scanner [45] and Avant-Guard [46] show a new attack (which is called data-to-control plane saturation attack) against SDN networks and provide solutions to prevent such attacks. Different from the previous work, this work is the first one to study the network topology visibility exploitation in the SDN network. Concurrently, SPHINX [47] proposes a unified approach to use network flow graphs to detect attacks that violate those learned flow graphs/modules. Different from their work, this work deeply investigates the vulnerabilities causing Network Topology Poisoning Attacks, as well as a low-overhead real-time defensive solution.

Related Poisoning Attacks in Legacy Networks. One notorious counterpart to the Host Lo-

cation Hijacking Attack is the ARP Cache Poisoning attack in Ethernet networks. That is, an adversary sends forged ARP messages to associate the IP address of the target host with the MAC address of a malicious host. By doing so, the adversary can hijack the entity of the target host, which is normally a gateway. However, the ARP Cache Poisoning attack has several differences from the Host Location Hijacking Attack as shown in Table 2.4. First, the attacking scope of the ARP Cache Poisoning is limited to a broadcast domain, i.e., the adversary must stay within the same broadcast domain with its target. By contrast, the adversary can launch the Host Location Hijacking Attack at any location of an OpenFlow network. Second, in addition to ARP reply packets, the Host Location Hijacking Attack can leverage almost all kinds of packets, e.g., ICMP echo, UDP and TCP, to usurp the location of the target host. In this point, the Host Location Hijacking Attack can be concealed in normal traffic to sidestep NIDS (Network Intrusion Detection Systems). Also from the defense perspective, the traditional mitigation strategies for ARP Cache Poisoning, such as the static ARP entry, may not be appropriate to apply directly to the SDN network since its static configuration undermines the dynamics handling capability of the OpenFlow network, e.g., tracking host migration between various OpenFlow access points [30]. To defend against the Host Location Hijacking Attack along with tracking network dynamics, in this work, we leverage OpenFlow specific capabilities to dynamically verify the host migration.

As illustrated in Table 2.5, an attack in legacy networks similar to the spirit of the Link Fabrication Attack is the STP Mangling (a.k.a, BPDU Falsification) attack [48], i.e., an adversary falsifies BPDUs with the smallest bridge ID to preempt the root of Spanning Tree. After faking the root, the adversary has potential to elaborate a Denial of Service or man-in-the-middle attack. However, the STP Mangling attack only disrupts the running of STP rather than injecting a fake link into network topology to poison the entire network operation. Also, some prior work about the exploitation of the view of network topology focus on only link-state routing protocols. Jones et al. [49] outline vulnerabilities of the design of OSPF and discuss the possible exploitations. Nakibly et al. [50] introduce two attacks to persistently falsify the topology of an OSPF network, which also incurs denial of service, eavesdropping and man-in-the-middle attacks. Such attacks

are launched by compromising the router entity or obtaining the pre-shared keys for the authentication of router. However, the Link Fabrication Attack in this work can be launched from the hosts residing in the data plane. Apart from a wired network, The link-state routing protocols in Mobile Ad Hoc networks, e.g., Optimized Link State Routing Protocol (OLSR), also incur similar security challenges. As mentioned in [51], an adversary can falsify links into OLSR topology by generating TC (or HNA) messages. Similar to OSPF Link Fabrication, OLSR Link Fabrication requires compromised routing entities, which is not required in our attacks. Another attack avenue in OLSR is the Wormhole attack [52, 53], which artificially creates wormholes in OLSR networks by recording traffic in one location and replaying it in another location. The OLSR Wormhole attack is only launched in a relay/replay manner. In contrast, our Link Injection attack can also be launched by falsifying the LLDP packets. In all, Table 2.5 summarizes the differences of those attack from the Link Fabrication Attack proposed in this work.

2.7 Discussion

Our attacks mainly focus on the data plane communication channel, i.e., an adversary can launch Link Fabrication Attack or Host Location Hijacking Attack on the top of compromised hosts. In fact, the security of OpenFlow control plane is also a security concern. As discussed in [54], most OpenFlow controllers and switch vendors lack full implementation of SSL/TLS. Seizing this security deficiency, an adversary can also launch man-in-the-middle attacks to manipulate control traffic between the controller and switches. We think that the message authentication can be extended to all OpenFlow messages to mitigate potential message falsification.

The fact that the OpenFlow controller handles all the layer 2 protocols on behalf of switches in OpenFlow networks leaves a room for other potential vulnerabilities from which the traditional network switches do not suffer. For instance, a new kind of Denial of Service attack [46], targeting at the *Packet-In* message handler, may saturate the control channel of OpenFlow as well as overload OpenFlow controllers. In order to systematically investigate the potential security issues, designing a new security fuzzer for SDN (controllers) may help us find more vulnerabilities, which is our future work.

2.8 Conclusion of This Work

In this work, we propose new SDN-specific attack vectors, Host Location Hijacking Attack and Link Fabrication Attack, which seriously challenge the core advantage of SDN, i.e., network-wide visibility. We demonstrate that the attacks can effectively poison the network topology information, thereby misleading the controller's core services and applications. We also systematically investigate the solution space and then present TOPOGUARD, a new security extension to the OpenFlow controllers, which provides automatic and real-time detection of Network Topology Poisoning Attacks. Finally, our prototype implementation shows a simple yet effective and efficient defense against the Network Topology Poisoning Attacks. In addition, we also released our prototype tool to help fix these vulnerabilities in widely used SDN controllers. This work contributes to enhance the security of network visibility in SDN.

3. CONGUARD: ENHANCING SECURITY OF CONCURRENT PROGRAMMING IN SDN*

3.1 Introduction

Software-Defined Networking (SDN) is rapidly changing the networking industry through a new paradigm of network programming, in which a logically centralized, programmable control plane, i.e., the *brain*, manages a collection of physical devices (i.e., the data plane). By separating data and control planes, SDN enables a wide range of new innovative applications from traffic engineering to data center virtualization, fine-grained access control, and so on [55].

Despite the popularity, unfortunately, SDN has also changed the attack surface of traditional networks. The network programmability provided by SDN controller introduces a list of network states such as host profile, switch liveness, link status, etc. By referencing proper network states, SDN controllers can enforce various network policies, such as end-to-end routing, network monitoring, and flow balancing. However, referencing network states is under the risk of introducing concurrency vulnerabilities because external network events can concurrently update the internal network states.

In this work, we study the security of SDN-provisioned network programmability in terms of concurrency programming model in widely-used SDN controllers. From the study, we find that network programmability in SDN is vulnerable to concurrency vulnerabilities, i.e., harmful race conditions, which can be exploited by the attackers to cause denial of services (e.g., controller crash, core service disruption) and privacy leakage, etc. Based on the harmful race conditions, we present a new attack, namely *state manipulation attack*, against the security and reliability of the SDN control plane. We note that this attack is closely tied to the unique SDN semantics, which makes all popular SDN controllers (e.g., Floodlight [25], ONOS [26], and OpenDaylight [27])

*Reprinted with permission from "Attacking the Brain: Races in the SDN Control Plane" by Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang and Guofei Gu, in Proceedings of the 26th USENIX Security Symposium, August 16-18, Vancouver, BC, Canada.

vulnerable.

In order to prevent harmful race conditions in SDN programs, this work aims to leverage the debugging technique to proactively locate and eliminate them before their exploitation by attackers. To achieve this, we face the following problems:

- First, how to locate race conditions in the SDN controller source code?
- Second, how to decide if a race condition is harmful or not?

For the first problem, the key challenge lies in that detecting race conditions in a program is generally undecidable. Although many data race detectors have been developed for different domains [56, 57, 58, 59, 60, 61], there is no existing tool to detect race conditions in the SDN controllers. We note that race conditions are different from data races but are a more general phenomenon; while data races concern whether accesses to shared variables are properly synchronized or not, race conditions concern about the memory effect of high-level races, regardless of synchronizations. Moreover, in SDN controllers there are many domain-specific happens-before rules. These rules must be properly modeled in a race detector; otherwise, a large number of false alarms will be reported. Therefore, conventional data race detectors are inadequate to find race conditions in SDN controllers.

To address the second problem, we develop a technique called *adversarial state racing* to detect harmful race conditions in the SDN control plane. Our key observation is that harmful race conditions are commonly rooted by two conflicting operations upon shared network states that are not commutative, i.e., mutating the scheduling order of them leads to a different state though the two operations can be well-synchronized (e.g., by using locks). Because there is no pre-defined order between the two conflicting operations, we can hence actively control the scheduler (e.g., by inserting delays) to run an adversarial schedule, which forces one operation to execute after another. If we observe an erroneous state (e.g., an exception or a crash) in the adversarial schedule, we have found a harmful race condition.

In this work, we have designed and implemented a framework called CONGUARD for detecting

harmful race conditions in the SDN control plane, and we have evaluated it on three mainstream open-source SDN controllers – Floodlight, ONOS, and OpenDaylight, with 34 applications in total. CONGUARD found 15 previously unknown harmful race conditions in these SDN controllers. We show that these harmful race conditions can incur serious reliability issues and remote attacks to the whole SDN network. Some attacks can be mounted by compromised hosts/virtual machines within the network, and some of them are possible if the SDN network uses in-band control messages[2] even when those messages are protected by SSL/TLS.

We highlight our key contributions as follows:

- We present a new attack on SDN networks by exploiting the harmful race conditions in the SDN control plane, which can greatly harzard network programmability in SDN.
- We design CONGUARD, a novel framework to pinpoint and manifest harmful race conditions in SDN controllers. We present a causality model that captures the domain-specific happens-before rules of SDN, which significantly increases the precision of race detection in the SDN control plane.
- We present an extensive evaluation of CONGUARD on three mainstream SDN controllers. CONGUARD has uncovered 15 previously unknown vulnerabilities that can result in both security and reliability issues. All these vulnerabilities were confirmed by the developers. We have already assisted the developers to patch 12 of them.

3.2 Background and Security Analysis

3.2.1 Background

The heart of SDN is a logically centralized control plane (i.e., SDN controllers) that is separated from the data plane (i.e., SDN switches). The programmable SDN controllers allow the network administrators to perform holistic management tasks, e.g., load-balancing, network visualization,

[2]There are two deployment options for SDN/OpenFlow networks, i.e., out-of-band option and in-band option. The out-of-band option requires a separated physical network for control traffic. In contrast, the in-band option allows OpenFlow switches also forward the SDN control traffic, which is a more convenient and cost-efficient way for large area networks [1, 62].

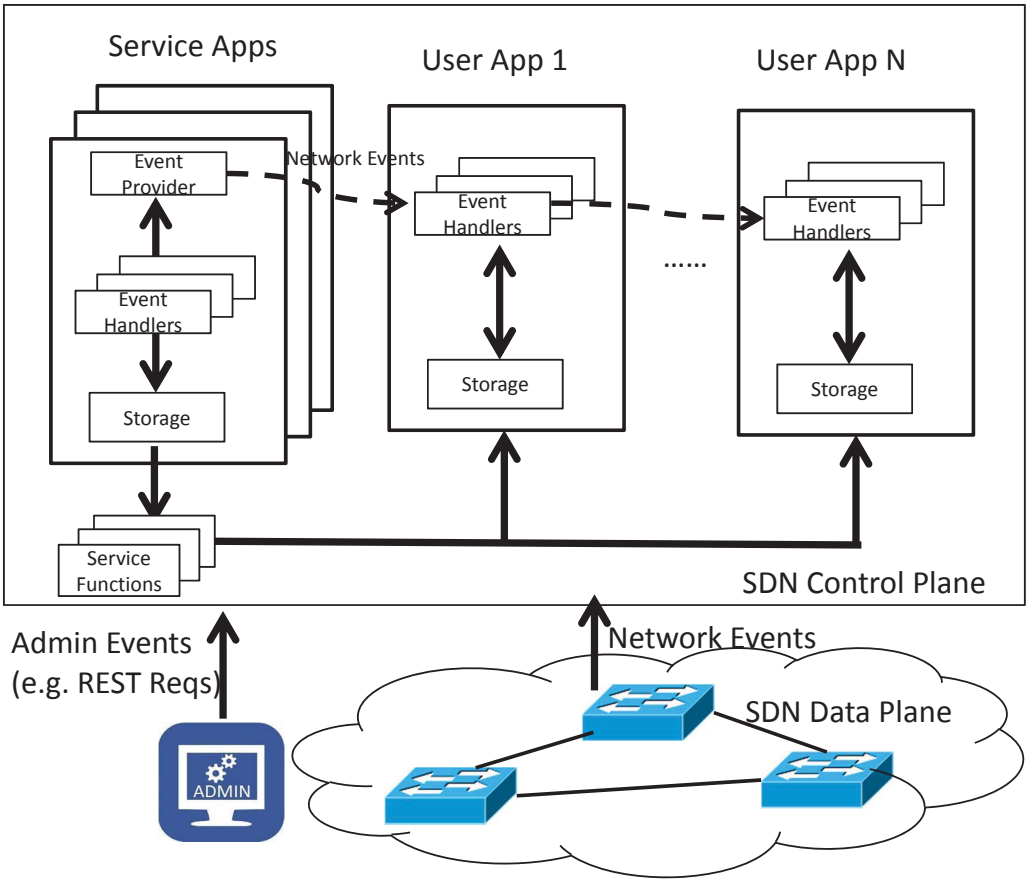


Figure 3.1: The abstraction model of the SDN control plane .

and access control. OpenFlow [1] is the dominant communication protocol between the SDN control plane and the data plane.

The SDN control plane embraces a concurrent modular programming model.

As shown in Figure 3.1, the SDN control plane embeds various modules (also known as applications) to enforce various network management policies, e.g., traffic engineering, virtualization, and access control. An SDN application manages a set of network states and provides *service functions* for other applications to reference the managed network states. For example, an access control application can install access control rules to all activated switches by querying the switch state from a switch manager application in the SDN controller. Also, each application operates in an event-driven fashion that implements handlers to process its corresponding events. It will

Table 3.1: Common network events in SDN controllers.

	Entity	Events
Network	HOST	JOIN, LEAVE
	SWITCH	JOIN, LEAVE
	PORT	UP, DOWN
	LINK	UP, DOWN
	OFP	PACKET_IN, OFP_PORT_STATUS, etc
Admin	REST	HOST_CONFIG, CREATE_VIP, etc

update its managed network states when it receives corresponding network events.

Also, some applications, namely *service applications*, in the SDN control plane paraphrase external network events (i.e., OpenFlow messages) to its own internal network events and dispatch them to other applications' event handlers. For example, when a switch manager application recognizes that a new OpenFlow-enabled switch[3] has joined the network, it issues a SWITCH_JOIN event to all corresponding handlers for policy enforcement. In addition, a network administrator can configure the SDN controller via REST APIs, which we call *administrative events* in the work.

Table 3.1 shows several network-related events and administrative events in the SDN control plane. In this work, we focus on these network events because they are commonly supported in all SDN controllers and they can be purposely generated by remote adversaries to exploit the race condition vulnerabilities.

We also note that certain events form implicit causal relationships. For example, a SWITCH_LEAVE event can implicitly trigger corresponding LINK_DOWN and HOST_LEAVE events. These implicit causal relationships must be captured to reason about race conditions in the SDN control plane. We present a comprehensive model of such causal relationships in Section 3.3.1.

3.2.2 Running Example of Harmful Race Conditions

Consider a real example of harmful race conditions we discovered in the Floodlight controller in Figure 3.2. When the controller receives a SWITCH_JOIN event, it updates a network state variable (i.e., *switches*) to store the profile of the joining switch. Shortly, the LinkDiscoveryMan-

[3]Without specific description, all term "switch" in this work refer to OpenFlow-enabled switch.

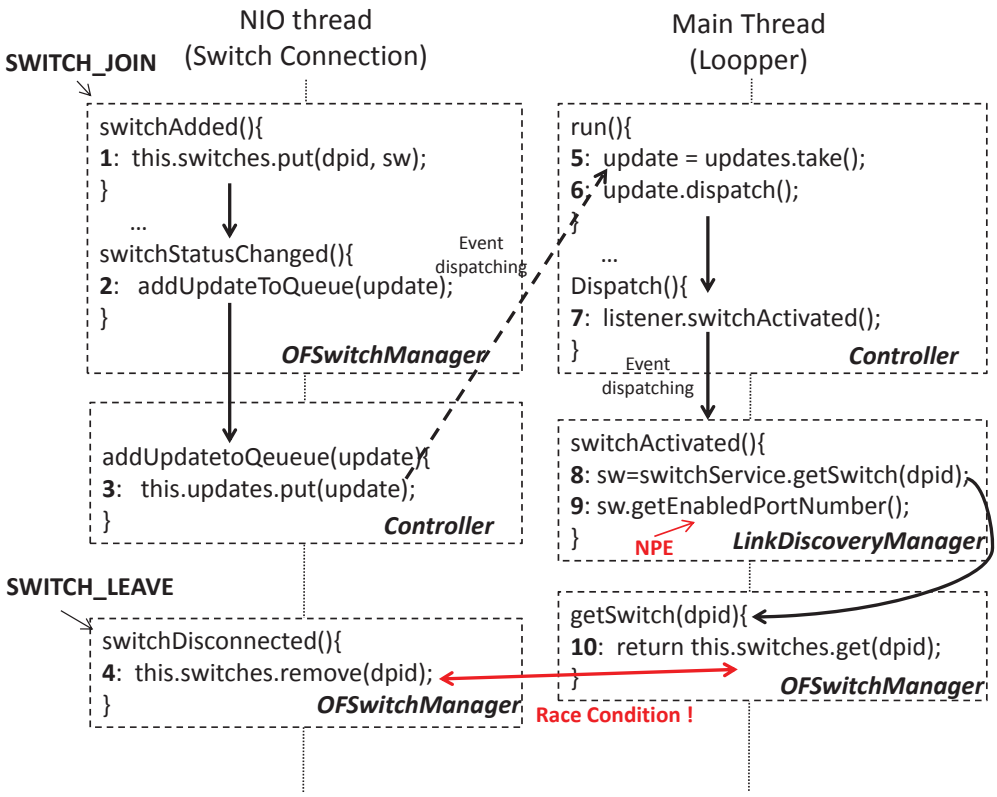


Figure 3.2: A harmful race condition in Floodlight v1.1.

ager application fetches the activated switch information from *switches* to discover links between switches. However, a `SWITCH_LEAVE` event can concurrently remove the profile of the activated switch in *switches*. If the operation at line 4 is executed before that at line 8, it will trigger a Null-Pointer Exception (NPE) when the null switch object is dereferenced at line 9, which leads to the crash of the thread and eventually causes Denial-of-Service (DoS) attacks on the controller.

The root cause of this vulnerability is a logic flaw in the implementation of Floodlight that permits a harmful race condition. In the SDN control plane, race conditions are common due to a massive number of network events on the shared network states. To meet the performance requirement, the event handlers in the SDN controller may run in parallel, which allows race conditions on the shared network states. By design, all such race conditions should be benign since they are protected by mutual exclusion synchronizations and do not break the consistency

of the network states. However, in practice, many of these race conditions become harmful races because it is difficult for the SDN developers to avoid logic flaws such as the one in Figure 3.2.

3.2.3 Threat Model

We consider two scenarios: non-adversarial and adversarial. In a non-adversarial case, a harmful race condition in the SDN control plane can happen rarely under normal network operation by asynchronous events as listed in Table 3.1. In contrast, in an adversarial case, the adversary could identify the harmful race conditions in the SDN controller source code and externally trigger them by controlling compromised hosts or virtual machines (e.g., via malware infection) with the system privilege to control network interfaces.

We do *not* assume that the adversary can compromise SDN controllers or switches, and we do *not* assume the adversary can compromise SDN applications or protocols. That is, we consider operating systems of SDN controllers and switches are well protected from the adversary, and the control channels between SDN controllers and SDN switches, as well as administrative management channels between administrators and SDN controllers, e.g., REST APIs, can be properly protected by SSL/TLS, which is particularly important when the SDN network is configured to use in-band control messages. Some of our attacks are possible even when the network is configured to use out-of-band control messages. For those attacks that assume in-band control messages, we assume control messages are properly protected by SSL/TLS.

Adversarial Event Generation. Host-related events (`HOST_JOIN`, `HOST_LEAVE`, and `OFP_PACKET_IN`) can be easily generated by an attacker from a compromised host or virtual machine without any knowledge about the switch. More specifically, to generate `HOST_JOIN` and `HOST_LEAVE` events, the attacker can simply enable/disable the network interface linked to a switch. The attacker can also send out crafted packets with randomized IP and MAC addresses to force a table miss in the switch's flow table[4], which can trigger `OFP_PACKET_IN` events. Switch port events (i.e., `PORT_UP` and `PORT_DOWN`) can also be indirectly generated by network

[4] An OpenFlow switch reports all packets to the SDN control plane if those packets do not hit its existing flow rule table.

interface manipulation (up and down) from a connected compromised host by using interface configuration tools, e.g., *ifconfig*.

In addition, an attacker can generate switch-dedicated events (i.e., `SWITCH_JOIN` and `SWITCH_LEAVE`) atop an in-band deployment of SDN networks. Even control messages are well protected by SSL/TLS, the attacker could still find important communication information (e.g., TCP header fields and types of control messages) between an SDN controller and switches by utilizing legacy techniques such as TCP/IP header analysis, size-based classification (given fixed size of control messages), etc. Then, the attacker may launch TCP session reset attacks [63] or drop control messages to disrupt the connection to generate `SWITCH_LEAVE`, thereby incurring `SWITCH_JOIN` subsequently. For example, as shown in Figure 3.3, we can use TCP reset to generate a `SWITCH_LEAVE` event in the Floodlight controller.

```
19:51:05.691 ERROR [n.f.c.i.OFChannelHandler:New I/O worker #11] Disconnecting switch
[00:00:00:00:00:00:00:01 from 192.168.1.102:59537] due to IO Error: Connection reset by peer
19:51:05.692 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 disconnected.
19:51:05.692 INFO [n.f.c.i.OFChannelHandler:New I/O worker #11] [[00:00:00:00:00:00:00:01 from
192.168.1.102:59537]] Disconnected connection
```

Figure 3.3: Switch leave event generated by TCP resets.

3.2.4 Exploitation of Harmful Race Conditions

To launch the attack, an adversary, who has no control of the SDN controller except sending external network events, first needs to figure out what external events to trigger a harmful race condition. For example, in Figure 3.2, a `SWITCH_JOIN` event can trigger a reference on the switch state and `SWITCH_LEAVE` event can trigger an update on the switch state. In addition, the attacker needs to trigger a "bad" schedule that can expose the harmful race condition. For example, a schedule in which the update on the switch state happens before the dereference.

Trigger Correlation. Since SDN controllers define different handler functions to process vari-

ous network events, we first statically analyze the program to extract a map from external events to their corresponding handler functions. Then, for each operation in a potentially harmful race condition, we backtrack the control flow graph from the operation to correlate the operation with the external event. In particular, we consider that a *trigger event* is correlated to a state reference operation and an *update event* is correlated to a state update operation. Moreover, we resolve potential contextual relations between *trigger event* and *state update event* by inspecting input parameters of state operations. For example, to exploit the vulnerability in Figure 3.2, the dpid of the *update event* `SWITCH_LEAVE` should be consistent with that of the *trigger event* `SWITCH_JOIN`.

Exploitation. In general, hitting a specific schedule that manifests harmful races is difficult because the space of all possible schedules is huge. Nevertheless, in SDN networks, an attacker can explore several effective ways to increase the chance of hitting an erroneous schedule.

First, we come up with a basic attack strategy, i.e., an attacker can repeat a proper sequence of crafted events (including ordered <*trigger event*, *update event*>). The trigger events will push the SDN controller to reference the state while the *update events* will modify the state. Hence, there are two resulting scenarios: 1) if the *update event* can update the network state before the reference happens, the exploitation succeeds; 2) if the *update event* falls behind the reference operation, a harmful race condition will not be triggered. In addition to injecting ordered attack event sequences, an attacker can probe the signals from SDN controllers to infer the attack results which can also benefit next-round exploitations. For example, in Figure 3.2, if the *update event* is late, we can observe the SDN controller send out LLDP packets to all enabled ports of the activated switch. The attacker can hence tune the timing interval between *trigger event* and *update event* to enhance the exploitability. Several other kinds of feedback information such as responses from service IP address and DHCP response/offer messages can also be utilized by the attacker to increase the success rate of the exploitations.

Moreover, an attacker can tactically increase the probability of success by selecting a larger vulnerable window [64] for a specific exploitation. The vulnerable window is the timing window that a concurrency vulnerability may occur. For some vulnerabilities, we found that their vulner-

able windows are subject to network conditions, e.g., the size of network topology or network round-trip latency. For example, as the harmful race condition in Figure 3.6, the attacker can launch the attack when the network delay is high. In such a case, an attacker can first utilize probe testing to pick up an advantageous condition to launch the attack.

Here, we discuss two attack cases exploiting harmful race conditions we detected in the LoadBalancer application of the Floodlight controller and DHCPRelay application of the ONOS controller.

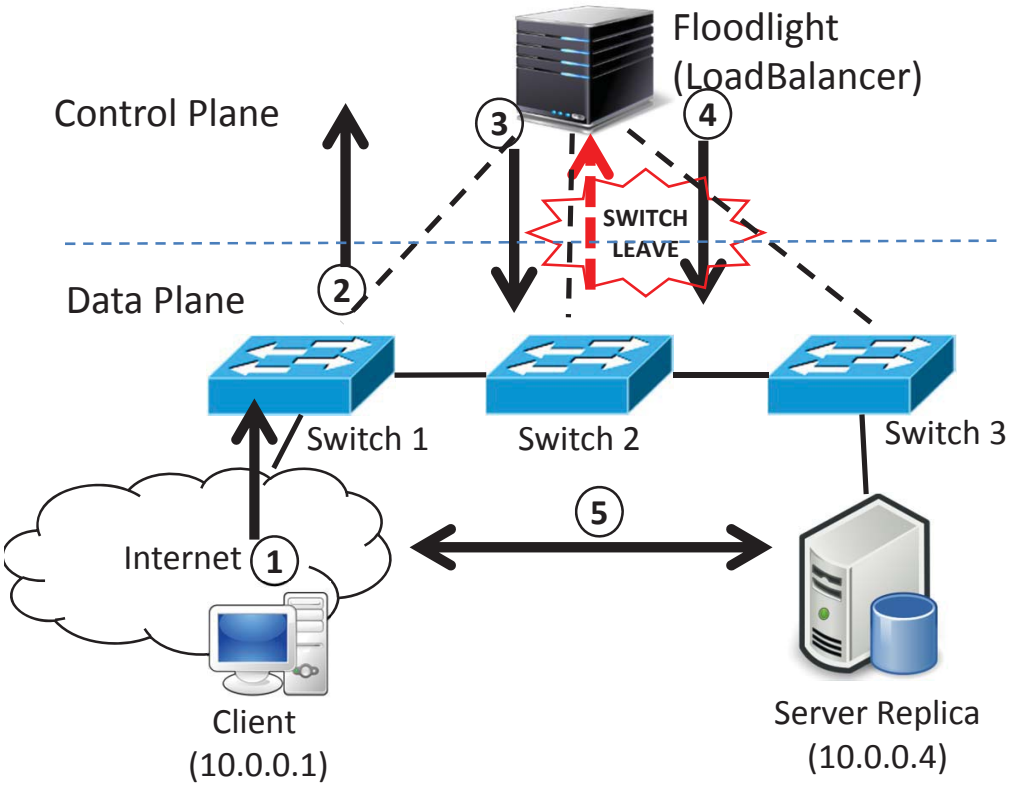


Figure 3.4: Attacking the Floodlight LoadBalancer application.

Stealing Privacy Information Figure 3.4 shows the workflow of the Floodlight LoadBalancer application. (1) A client sends out a service request packet with the virtual IP address

(10.10.10.10) of server. (2) Switch 1 issues an OFP_PACKET_IN event to Floodlight controller to report a table-miss packet. (3) The OFP_PACKET_IN handler selects a service replica (10.0.0.4) to process the request and installs inbound flow rules in each switch along the route from the client to the replica. Besides, for routing and privacy purposes, an extra flow rule is installed into switch 1 to convert the destination IP address of packets from virtual IP address (10.10.10.10) to physical IP address of the replica (10.0.0.4). (4) The OFP_PACKET_IN handler also installs outbound flow rules from the service replica to the client and restores the virtual IP address on Switch 1 (i.e., from 10.0.0.4 to 10.10.10.10). (5) As a result, the client can successfully communicate with the server replica.

We found a harmful race condition in this application, i.e., a concurrent SWITCH_LEAVE event from any switch along the routing path can trigger an internal exception of the Floodlight controller and further violate the policy enforcement from step (3) to step (4). If that happens, no source IP address conversion rule (from 10.10.10.10 to 10.0.0.4) will be installed in switch 1. As a result, the sensitive physical IP address information is disclosed to the client which sent requests to the public service.

In order to exploit the harmful race condition remotely, we set up an experiment, as shown in Figure 3.4 in Mininet [31]. To launch the attack, we periodically injected OFP_PACKET_IN and SWITCH_LEAVE events. In particular, we updated the source IP address of a host and sent out ICMP echo requests (with the destination IP address of the public service 10.10.10.10) into the network to trigger the OFP_PACKET_IN messages. We also reset the TCP session between switch 2 and the Floodlight controller to generate SWITCH_LEAVE. As long as observing an ICMP echo reply whose source IP address is the physical replica (10.0.0.4), we consider the exploitation succeeds. Consequently, we successfully sniffed the physical IP address of the service replica after injecting tens of SWITCH_LEAVE events, as shown in Figure 3.5 below.

Disrupting Packet Processing Service. In order to provide a DHCP service in different subnets, the DHCPRelay application in the ONOS controller relays DHCP messages between DHCP clients and the DHCP server. However, due to a harmful race condition, a conflicting

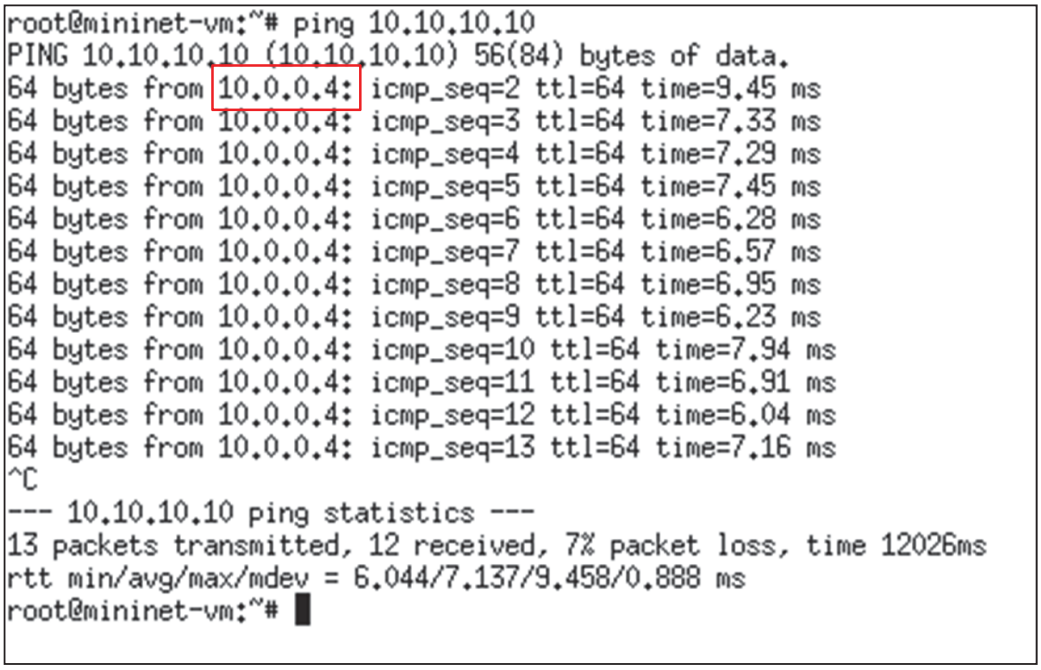


Figure 3.5: Privacy leakage in the Floodlight LoadBalancer application.

HOST_LEAVE event can manipulate the internal state of the host, which may result in an unexpected exception and further disrupt the packet processing service when the DHCPRelay application relays DHCP response/offer messages to the sender, as illustrated in Figure 3.6. The root cause of this vulnerability lies in that the host state variable referenced by DHCPRelay application can be nullified by a HOST_LEAVE event.

We set up an attack experiment in Mininet (with 500ms delay link between the DHCP server and its connected switch), where we injected ordered attack event sequences, i.e., <OFP_PACKET_IN, HOST_LEAVE>. In detail, we controlled a host to send out a DHCP request (to generate OFP_PACKET_IN) and turn off the network interface (to inject a HOST_LEAVE event) immediately after the transmission of the DHCP request. As a result, the harmful race condition is triggered by injecting an attack event sequence, which actually disrupts the packet processing service (as shown in Figure 3.7) to dispatch the incoming packets to OFP_PACKET_IN event handlers of SDN controller/applications. The exploitation possibility of such harmful race condition is comparatively high for a remote attacker since its vulnerable window is subject to round-trip delay between the

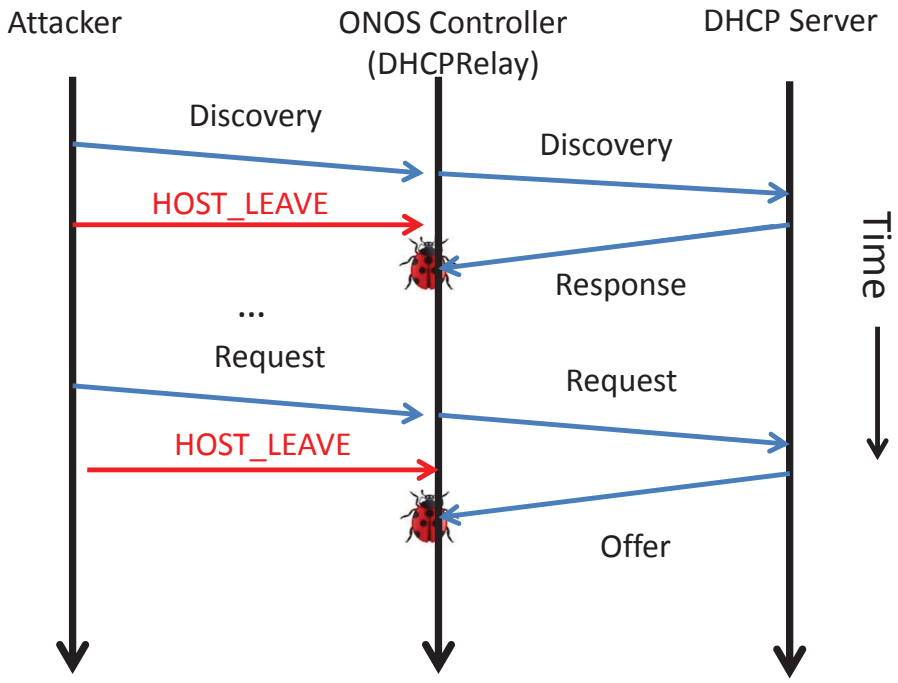


Figure 3.6: Attacking the ONOS DHCPRelay application.

ONOS controller and the DHCP server. In this case, a tactical attacker can even pick up a network congestion timing to increase the success ratio of the exploitation.

```
WARN  | ew I/O worker #2 | PacketManager   | 76 - org.onosproject.onos-core-net - 1.7.2.SNAPSHOT | Packet
processor org.onosproject.dhcprelay.DhcpRelay$DhcpRelayPacketProcessor
@6018f73a threw an exceptionjava.lang.NullPointerException
at org.onosproject.dhcprelay.DhcpRelay$DhcpRelayPacketProcessor.sendReply(DhcpRelay.java:391)
[172:org.onosproject.onos-app-dhcprelay:1.7.2.SNAPSHOT]
at org.onosproject.dhcprelay.DhcpRelay$DhcpRelayPacketProcessor.processDhcpPacket(DhcpRelay.java:333)
[172:org.onosproject.onos-app-dhcprelay:1.7.2.SNAPSHOT]
```

Figure 3.7: Service disruption in the ONOS DHCPRelay application.

3.2.5 Research Challenges

In order to prevent harmful race conditions in SDN programs, we aim to leverage the debugging technique to locate and eliminate them before their exploitation by attackers proactively. To locate harmful race conditions, our basic idea is to use dynamic analysis to first detect a superset of potentially harmful race conditions, and then use adversarial state racing to manifest those real harmful ones. More specifically, given a target SDN controller, we first analyze its dynamic behavior (by generating network events as inputs to it and then tracing the execution) to detect race conditions consisting of two race operations on a shared network state. These two operations may or may not have a common lock protecting them, but there should not be any predefined order causality between them. Then, for each pair of such operations, we re-run the SDN controller but force it to follow an erroneous schedule to check if a race condition is harmful or not.

In this step, there are two major challenges:

- *First, how to avoid reporting a myriad of race warnings that are in fact false alarms?* Lack of accurate modeling of the SDN semantics can significantly impede the precision of race detection. For example, in Figure 3.2, without reasoning the causality order between line 3 and line 5 for the internal event dispatching, the state update operation at line 1 and state reference at line 10 will be reported as a false positive.

- *Second, how to manifest and verify harmful race conditions?* Witnessing/reproducing concurrency errors is infamously difficult since they may be non-deterministic that only occur in rare scenarios with the special input and schedule. For example, the vulnerability in Figure 3.2 is triggered when the write operation on the state variable *switches* (e.g., triggered by the `SWITCH_JOIN` event) occurs before the read operation of the state variable (e.g., caused by the `SWITCH_JOIN` event). In addition, the runtime context of the two state operations must be consistent, e.g., the value of *dpid* at lines 4 and 10 must be equal.

To address the first challenge, we develop an execution model of the SDN control plane that formulates happens-before semantics in the SDN domain, which can help us greatly reduce false

positives. For the second challenge, we develop an adversarial testing approach with a context-aware and deterministic scheduling technique, called *Active Scheduling*, to verify and manifest harmful race conditions.

3.3 System Design

In this section, we present our framework, CONGUARD, for detecting and validating the race condition vulnerabilities in SDN controllers. CONGUARD contains two main phases: (i) locating race conditions in the controller source code by utilizing dynamic analysis , (ii) pinpointing harmful race conditions from located race operations by using adversarial state racing.

3.3.1 Modeling the SDN Control Plane

Generally, an execution of an SDN controller corresponds to a sequence of operations performed by threads on a collection of state objects. For detecting races, we would like to develop a model such that it captures all the critical operations inside the SDN control plane (as an execution trace) and their causality relationships in any execution of the SDN controller (as happens-before relations). Different from general multi-thread programs, there are a number of distinct types of operations and domain-specific causality rules in the SDN control plane.

Execution Trace: First, we model an execution of the SDN control plane as a sequence of operations as listed following:

- ***read(T,V)***: reads variable *V* in thread *T*.
- ***write(T,V)***: writes variable *V* in thread *T*.
- ***init(A)***: initializes the functions of application *A* in the SDN control plane.
- ***terminate(A)***: terminates the functions of application *A* in the SDN control plane.
- ***dispatch(E)***: issues event *E*.
- ***receive(H,E)***: receives event *E* by event handler *H*.
- ***schedule(TA)***: instantiates a singleton task *TA*.
- ***end(TA)***: terminates a singleton task *TA*.

Happens-Before Causality: In this work, we utilize happens-before relations [65] to model the concurrency semantics of the SDN controller. A happens-before relation is a transitively closed binary relation to represent *order causality* between two operations, as denoted by $\prec$ in this work. That is, $\alpha \prec \beta$ means operation α happens before operation β. Moreover, we utilize $\alpha <_\tau \beta$ to denote that operation α occurs before operation β in an execution trace τ. As illustrated in Figure 3.8, we list happens-before relations we derive in the SDN context by studying implementations of SDN controllers and OpenFlow switch specification [66]. For simplicity, we do not list those happens-before rules widely used in traditional thread-based programs, e.g., program order rules and fork/join rules. Instead, we elaborate some happens-before rules mostly unique to the SDN control plane as listed in Figure 3.8, which we intend to expand over time.

Application Life Cycle. We define two happens-before rules to model the life cycle of an SDN application. First, an application must be initialized before it can handle any network event; second, all event handling operations in an application must happen before the deactivation of the application.

Event Dispatching. For each network event (as shown in Table 3.1), we consider dispatching of the event must happen before the receipt of the event in various event handlers.

Sequential Event Handling. Moreover, most SDN controllers (e.g., OpenDaylight, ONOS, Floodlight, Pox, Ryu, etc.) handle network events sequentially, i.e., at any time an event can only be processed in a single event handler. Hence, we deduce that the receipt of a specific event for different handler functions should follow their orders in the observed execution trace.

Switch Event Dispatching. Before issuing `SWITCH_JOIN` event, the SDN control plane must receive an `OFP_FEATURES_REPLY` event that includes important information of the joining switch, e.g., *Datapath ID*.

Port Event Dispatching. The SDN control plane monitors `OFP_PORT_STATUS` OpenFlow messages to detect the addition and deletion of switch ports in the data plane. Consequently, the corresponding *PortManager* application dispatches `PORT_UP` or `PORT_DOWN` events to inform other applications.

Implicit Host Leave or Link Down. In the SDN control plane, we also monitor implicit causalities between events, i.e., a `PORT_DOWN` or `SWITCH_LEAVE` event may implicitly indicate a `HOST_LEAVE` or `LINK_DOWN` event.

Singleton Task. We note that a specific singleton task can only be instantiated once at a time. In order to avoid non-determinism of thread scheduling (especially in a thread pool), we define one happens-before relation to model the causality order that the last completion of a specific singleton task happens before the next schedule of the task.

3.3.2 Detecting Race State Operations

Our algorithm for detecting race state operations upon shared network state variables is based on the happens-before rules constructed in the previous section. Given an observed execution trace τ of an SDN controller, we construct happens-before relations $\prec$ between each pair of operations listed in the execution model in Section 3.3.1. For each pair of memory access operations, i.e., (α, β), on the same state variable, we report (α, β) as a race state operation, if it meets two conditions: 1) either α or β updates the state variable; 2) $\alpha \not\prec \beta$ and $\beta \not\prec \alpha$.

Taking the raw execution trace as input, we first conduct an effective preprocessing step to filter out redundant operations in the trace. Specifically, we remove those operations on thread-local or immutable data, since we only need to reason about conflicting operations on shared state variables. We also perform a duplication checking to prune duplicated *write* and *read* operations. In SDN, an event handler can repeatedly process identical network events, which produces a large number of duplicated events in the trace. Removing such redundant events significantly improves the efficiency of race condition detection.

We note that standard vector-clock based techniques [59] for computing happens-before relation is difficult to scale to the SDN domain, which typically contains a large number of network events and threads. Instead, we develop a graph-based algorithm [67, 60] that constructs a directed acyclic graph (DAG) from the preprocessed trace to detect commutative races. In the DAG, nodes denote operations, and edges denote happens-before relations between them. The rationale is that the problem of checking happens-before can be converted to a graph reachability problem. To fa-

Application Life Cycle

$$\frac{\alpha \in init(A) \qquad \beta.app\_id = A.app\_id}{\alpha \prec \beta}$$

$$\frac{\alpha.app\_id = A.app\_id \quad \beta \in terminate(A)}{\alpha \prec \beta}$$

Event Dispatching

$$\frac{\alpha \in dispatch(E) \quad \beta \in receive(H,E)}{\alpha \prec \beta}$$

Sequential Event Handling

$$\frac{\alpha = receive(H_1,E) \quad \beta = receive(H_2,E) \quad \alpha <_\tau \beta}{\alpha \prec \beta}$$

Switch Event Dispatching

$$\frac{\begin{array}{c}\alpha = receive(H,E_1) \quad \beta = dispatch(E_2) \\ E_1.type = \texttt{OFP\_FEATURES\_REPLY} \quad E_2.type = \texttt{SWITCH\_JOIN} \\ E_1.switch\_id = E_2.switch\_id\end{array}}{\alpha \prec \beta}$$

Port Event Dispatching

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \\ E_1.type = \texttt{OFP\_PORT\_STATUS} \quad E_2.type = \texttt{PORT\_UP} \\ E_1.port\_id = E_2.port\_id \quad E_1.reason = \texttt{OFPPR\_ADD}\end{array}}{\alpha \prec \beta}$$

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \\ E_1.type = \texttt{OFP\_PORT\_STATUS} \quad E_2.type = \texttt{PORT\_DOWN} \\ E_1.port\_id = E_2.port\_id \quad E_1.reason = \texttt{OFPPR\_DELETE}\end{array}}{\alpha \prec \beta}$$

Explicit Link Down and Host Leave

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \quad E_1.port\_id = E_2.port\_id \\ E_1.type = \texttt{PORT\_DOWN} \quad E_1.type = \{\texttt{LINK\_DOWN},\texttt{HOST\_LEAVE}\} \\ E_1.port\_id = E_2.port\_id\end{array}}{\alpha \prec \beta}$$

$$\frac{\begin{array}{c}\alpha = (H,E_1) \quad \beta = dispatch(E_2) \quad E_1.switch\_id = E_2.switch\_id \\ E_1.type = \texttt{SWITCH\_LEAVE} \quad E_1.type = \{\texttt{LINK\_DOWN},\texttt{HOST\_LEAVE}\}\end{array}}{\alpha \prec \beta}$$

Singleton Task

$$\frac{\alpha = end(TA) \quad \beta = schedule(TA) \quad \alpha <_\tau \beta}{\alpha \prec \beta}$$

Figure 3.8: Happens-before rules in the SDN control plane.

cilitate race detection, we group operations by their accessed state variable. We can then pinpoint race operations by checking if there is a path between each pair of conflicting nodes in the DAG.

Specifically, if a *write* node and a *read* node are from the same group, and there is no path between them, we report they are race operations.

3.3.3 Adversarial State Racing

Verifying a potentially harmful race condition is a challenging problem because it can only be triggered in a specific execution branch of the SDN controller under a certain schedule of operations. An intuitive approach is to instrument control logic to force an erroneous execution order, e.g., the state update executes before the state reference. However, we find such strawman approach introduces non-determinism due to two reasons. First, SDN applications may reference the same network state variable in different program branches. Second, inconsistent input parameters of the library methods upon a state variable may impede the verification, e.g., scheduling *switches.remove(sw1)* before *switches.get(sw2)* will not lead to a harmful race condition. To address the first problem, we propose to explore all possible program branches to the reference operation upon the state variable and verify all of them at runtime deterministically. To address the second problem, we check the consistency of parameters for library methods upon the same state variable.

Active Scheduling. Taking a potentially harmful race condition as input, our active scheduling technique re-executes the program to force two operations (like operations in line 4 and line 10 in Figure 3.2) to follow a specific erroneous order, as shown in Figure 3.9. To force the deterministic schedule in a certain control branch (and external triggers), we put an exclusive waypoint (a check point in the code) to differentiate it with other branches. In addition to utilizing the waypoint to ensure execution context, we also add four atomic control points (P1, P2, P3, and P4) and one flag (F1) to enforce the deterministic scheduling between the state reference operation and the state update operation with consistent runtime information.

More specifically, we place P1 ahead of Operation 1, P2 ahead of Operation 2, P3 after Operation 1 and P4 after Operation 2. The active scheduling works as follows: In P1, if the corresponding waypoint is marked (which means the branch under test is covered), we pause Thread *a* by using a blocking method and save the runtime parameter value if necessary (e.g., the *dpid* of

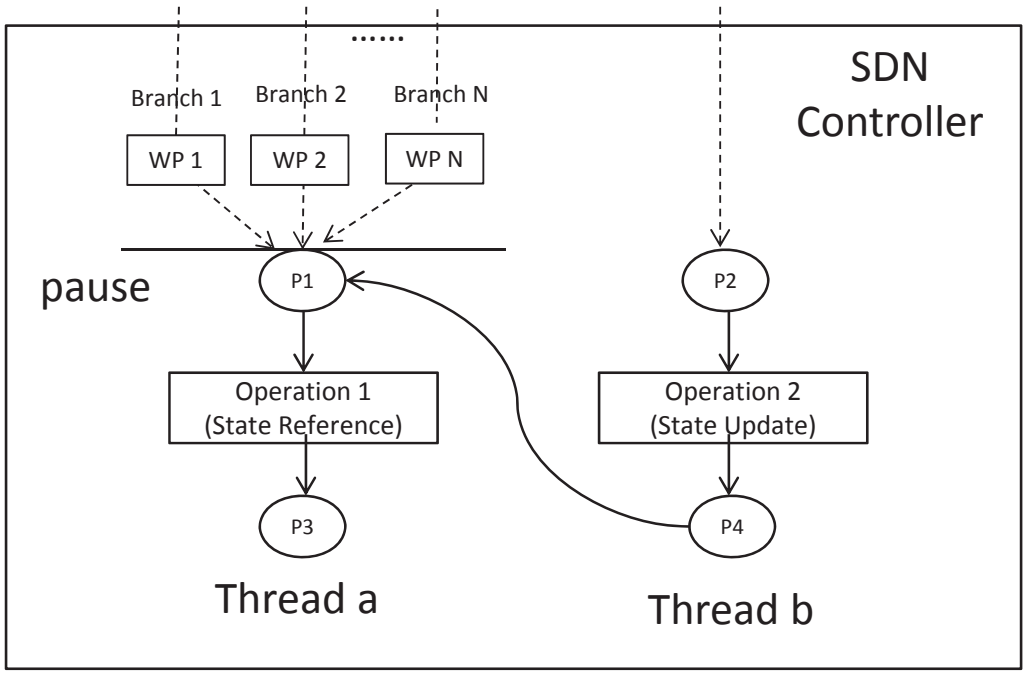


Figure 3.9: The workflow of Active Scheduling Scheme.

switches.getSwitch(dpid) in Figure 3.2). When Thread *b* enters P2, we set flag F1 if two conditions are satisfied: (1) Thread *a* is blocked; (2) the runtime value for Operation 2 is equal to runtime value of Operation 1. In P4, we unblock Thread *a* if flag F1 is set.

3.4 System Implementation

We have implemented and tested it on three mainstream SDN controllers, including Floodlight [25], ONOS [26] and OpenDaylight [27].

Input Generation: To inject network events, we introduce an SDN control plane specific input generator in our framework. We utilize Mininet 2.2 [31], an SDN network simulator, to mock an SDN testbed. Mininet can generate all the network events as shown in Table 3.1. In addition, we create test scripts to send REST requests as another source of inputs to the SDN controller.

Instrumentation: We use the ASM [68] bytecode rewriting framework to instrument and analyze SDN controllers at the Java bytecode level. For each event in the execution trace, we assign a global incremental number as its identifier, a location ID to store its source code context (i.e.,

class name and line number), and a thread ID. At runtime, the execution traces and contextual metadata are stored in a database (H2 [69]). Since we focus on locating harmful race conditions in the SDN controller source code, we exclude external packages in third-party libraries from the instrumentation. In addition, to improve performance, we only instrument those network state variables with reference data types and exclude primitive types (e.g., int, bool) because typically only reference types are involved in harmful race conditions.

We log memory accesses (e.g., *putfield* and *getfield*) upon objects and class fields as well as their values as metadata. We note that the SDN control plane embraces heterogeneous storages for network state including third party libraries such as java.util.HashMap. Failing to resolve those storage methods (e.g., *remove()* and *get()*) would lead to missing of potential vulnerabilities. Hence, we map those library method invocation operations as *write* or *read* operations upon the state object. For example, we consider *switches.remove(dpid)* is a *write* operation on *switches*.

We locate two kinds of event dispatching manners in SDN controllers, i.e., queue-based and observer-based. For queue-based rules, we record write and read operations upon global event queues as *dispatch* and *receive* operations. In contrast, for observer-based scheme, we log the invocations of event handler functions with the context of application name as *receive* operations upon the event.

We track *schedule* and *end* task operations by monitoring the life-cycle of *run()* method for singleton tasks. We log application life-cycle operations (i.e., *init* and *terminate*) by monitoring application-related callback methods (as listed in Table 3.2) with the identifier of the name of the class.

Table 3.2: Initialization and destroy methods of SDN controllers.

Controller	**Init Methods**	**Destroy Methods**
Floodlight	init(), startup()	–
ONOS	activate()	deactivate()
OpenDaylight	init()	destroy()

Active Scheduling: We implement active scheduling as a service module in the SDN controller that provides functions such as atomic control points (i.e., P1-P4) and waypoints. In order to cover all potential branches to trigger the bug, we statically generate the call graph of the tested controller. For each race state operations, we backtrack all paths (i.e., sequences of calling methods) to reach the state reference operation. For each path, we choose the method as the waypoint if it is: (1) nearest to the use operation in the call graph and (2) not listed in any other path. Taking the location of race state operations and all its corresponding waypoints as input, we instrument the SDN controller to invoke methods of the active scheduling service module.

3.5 Evaluation

In this section, we present our evaluation results of on the three mainstream open-source SDN controllers with 34 applications as listed in Table 3.3. We hosted all the tested SDN controllers on a machine running GNU/Linux Ubuntu 14.04 LTS with dual-core 3.00 GHz CPU and 8 GB memory.

3.5.1 Detection Results

Table 3.4 summarizes our race detection results in Floodlight 1.1 and 1.2, ONOS 1.2 and OpenDaylight 0.1.7. In total, our tool found 153 race conditions on 22 network state variables in Floodlight 1.1, 184 race conditions on 35 variables in Floodlight 1.2, 221 race conditions on 26 variables in OpenDaylight, and 13 race conditions on 5 variables in ONOS. The numbers of detected race operations and network state variables in ONOS are much smaller than those of the other two controllers, because ONOS uses a centralized data storage to manage the network states. In addition, our results show that our offline trace analysis is highly effective and efficient. The preprocessing step reduces the size of traces (by removing redundant events) by more than 87%. For all the three controllers, the offline analysis was able to finish in less than two minutes.

To evaluate the effectiveness of the SDN domain-specific happens-before rules, we compared the following two configurations on running race detection of with Floodlight version 1.1: (1) enforces only thread-based happens-before rules; (2) enforces both thread-based and SDN-specific

Table 3.3: Tested SDN applications

Controller	Application Name	Location
Floodlight	Switch Manager	net.floodlightcontroller.core.internal
	Link Manager	net.floodlightcontroller.linkdiscovery
	Host Manager	net.floodlightcontroller.devicemanager
	Topology Manager	net.floodlightcontroller.topology
	Forwarding	net.floodlightcontroller.forwarding
	LoadBalancer	net.floodlightcontroller.loadbalancer
	Firewall	net.floodlightcontroller.firewall
	DHCP Server	net.floodlightcontroller.dhcpserver
	AccessControlList	net.floodlightcontroller.accesscontrollist
	Static Route Pusher	net.floodlightcontroller.staticflowentry
	Statistics	net.floodlightcontroller.statistics
OpenDaylight	Switch Manager	org.opendaylight.controller.switchmanager
	Statistics Manager	org.opendaylight.controller.statisticsmanager
	Topology Manager	org.opendaylight.controller.topologymanager
	ForwardingRulesManager	org.opendaylight.controller.forwardingrulesmanager
	HostTracker	org.opendaylight.controller.hosttracker
	ArpHandler	org.opendaylight.controller.arphandler
	LoadBalancerService	org.opendaylight.controller.samples.loadbalancer
	SimpleForwardingImpl	org.opendaylight.controller.samples.simpleforwarding
	Static Routing	org.opendaylight.controller.forwarding.staticrouting
ONOS	OpenFlow Controller	org.onosproject.openflow.controller.impl
	Switch Manager	org.onosproject.store.device.impl
	Host Manager	org.onosproject.store.host.impl
	Packet Manager	org.onosproject.store.packet.impl
	Link Manager	org.onosproject.store.link.impl
	ProxyArp	org.onosproject.proxyarp
	ReactiveForwarding	org.onosproject.fwd
	HostMobility	org.onosproject.mobility
	SegmentRouting	org.onosproject.segmentrouting
	ACL	org.onosproject.acl
	DHCP	org.onosproject.dhcp
	DHCPRelay	org.onosproject.dhcprelay
	FaultManagement	org.onosproject.faultmanagement
	FlowAnalyzer	org.onosproject.flowanalyzer

rules. Our results show that adopting SDN-specific happens-before rules reduces 105 reported race conditions in total (153 vs 258). We manually inspected all those race condition warnings filtered by SDN-specific rules and found that all of them are false positives. We expect that the happens-before rules formulated in this work greatly complement existing thread-based rules for

Table 3.4: Overall race detection results.

1	2	3	4	5	6	7	8
SDN Controller		Trace Processing			Race Detection Results		
Name	Version	#RT	#OT	RE	OTATime	#Races	#RSVs
Floodlight	1.1	234,517	8,063	96.6%	43s	153	22
	1.2	410,128	52,271	87.2%	101s	184	35
OpenDaylight	0.1.7	47,855	3,752	92.1%	5s	221	26
ONOS	1.2	69,214	1,292	98.1%	5s	13	5

conducting more precise concurrency defect detection in SDN controllers.

3.5.2 Comparing With Existing Techniques

To evaluate the effectiveness of our approach for identifying harmful race conditions, we also compared with an SDN-specific race detector, SDNRacer [56], and a state-of-the-art general dynamic race detector, RV-Predict (version 1.7) [58].

Comparing with SDNRacer. SDNRacer is a dynamic race detector that also locates concurrency violations in SDN networks. Because SDNRacer can also work on the Floodlight controller, we directly compared their results with ours. In a single-switch topology, SDNRacer reported 2, 281 data races. However, we find that none of those data races are relevant to our detected harmful race conditions. The reason lies in that SDNRacer only models memory operations in SDN *switches* but ignores internal state operations in SDN controllers. In this sense, we consider our new detection solution is orthogonal and complementary to SDNRacer.

Comparing with RV-Predict. RV-Predict is the state-of-the-art general-purpose data race detector that achieves maximal detection capability based on a program trace but does not consider harmful race conditions, and does not have SDN-specific causality rules. We evaluated RV-Predict as a Java agent for Floodlight v1.1 with our implemented network event generator and REST test scripts. We found that RV-Predict reported a total of 29 data races. However, none of them was harmful and none of them was related to harmful race conditions[5]. The reason is that all those harmful race conditions are caused by well-synchronized operations in Java concurrent libraries,

[5] We manually backtracked the call graph information for every data race reported by RV-Predict and checked if it could lead to harmful race conditions.

which are not data races.

3.5.3 Runtime Performance

We evaluated the runtime performance of for trace collection using Cbench [70], an SDN controller performance benchmark. We use Cbench to generate a sequence of `OFP_PACKET_IN` events and test the delay. To remove network latency, we locate Cbench in the same physical machine with SDN controllers and range testbed from 2 switches to 16 switches. Our results show that incurs about 30X, 10X and 8X latency overhead for Floodlight, ONOS and OpenDaylight, respectively. The network functionalities can work properly and the instrumentation does not affect the collection of execution traces. The performance overhead mainly comes from instrumentation sites that frequently write event traces into the database. Although apparently 8X-30X latency is not small, we note that our tool is for *offline* bug/vulnerability finding purpose in the development and testing phase instead of online use in the actual operation phase. Thus, the overhead is acceptable as long as the tool can effectively find true bugs/vulnerabilities.

```
10:30:58.430 ERROR [n.f.c.i.Controller:main] Exception in controller updates loop
java.lang.NullPointerException: null
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.generateLLDPMessage(L
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.sendDiscoveryMessage(
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.discover(LinkDiscoveryM
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.processNewPort(LinkDis
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.switchActivated(LinkDis
at net.floodlightcontroller.core.internal.OFSwitchManager$SwitchUpdate.dispatch(OFSwitchMa
```

Figure 3.10: A harmful race condition causes the Floodlight controller out of service.

3.5.4 Impact Analysis of the Detected Vulnerabilities

By utilizing adversarial testing, we identified 15 concurrency bugs/vulnerabilities caused by harmful race conditions including 10, 2, 3 in Floodlight, ONOS and OpenDaylight, respectively. Furthermore, we conduct an impact analysis for those vulnerabilities, as shown in Table 3.5. We

Table 3.5: Summary of uncovered harmful race conditions.

Controller	Application	Bug#	Correlated Attack Event Pairs <trigger event, update event>	Impact Vector			
				#1	#2	#3	#4
Floodlight	Link Discovery Manager	1*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●		●	
		2*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●		●	
		3*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●		●	
	DHCPServer	4*	<SWITCH_JOIN, SWITCH_LEAVE>, <PORT_UP, SWITCH_LEAVE>	●			
	Load Balancer	5*	<OFP_PACKET_IN, SWITCH_LEAVE>		●	●	●
		6*	<OFP_PACKET_IN, SWITCH_LEAVE>		●	●	●
		7†	<OFP_PACKET_IN, REST_REQUEST>		●	●	●
		8†	<OFP_PACKET_IN, REST_REQUEST>		●	●	●
		9†	<OFP_PACKET_IN, REST_REQUEST>		●	●	
	Statistics	10†	<REST_REQUEST, SWITCH_LEAVE>			●	
ONOS	SegmentRouting	11	<OFP_PACKET_IN, HOST_LEAVE>		●	●	●
	DHCPRelay	12	<OFP_PACKET_IN, HOST_LEAVE>		●	●	●
OpenDaylight	Host Tracker	13†	<REST_REQUEST, HOST_LEAVE>			●	
		14	<HOST_JOIN, HOST_LEAVE>				●
	Web UI	15†*	<REST_REQUEST, SWITCH_LEAVE>			●	

* exploitable if the network is configured with in-band control, or if the adversary has access to the out-of-band network

† exploitable if the adversary can send authenticated administrative events (REST APIs) to the controller

```
22:33:28.298 ERROR [n.f.c.i.OFChannelHandler:New I/O worker #12]
Error while processing message from switch [00:00:00:00:00:00:00:01 from 192.168.1.102:5281
state net.floodlightcontroller.core.internal.OFChannelHandler$CompleteState@32250656
java.lang.NullPointerException: null
at net.floodlightcontroller.loadbalancer.LoadBalancer.processPacketIn(LoadBalancer.java:234) ~
...
at java.lang.Thread.run(Thread.java:745) [na:1.7.0_79]22:33:28.299
WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 disconnected.
```

Figure 3.11: A harmful race condition in Floodlight causes disconnection of a switch.

note that a single harmful race condition can have multiple impacts depending on different program branches/schedules and contexts.

Impact #1: System Crash. In Floodlight, we found 4 serious crash bugs, in which three of them (**Bug-1**, **Bug-2** and **Bug-3**) are in the LinkDiscoveryManager application and one of them (**Bug-4**) is in DHCPSwitchServer application. We manifested such vulnerabilities by active

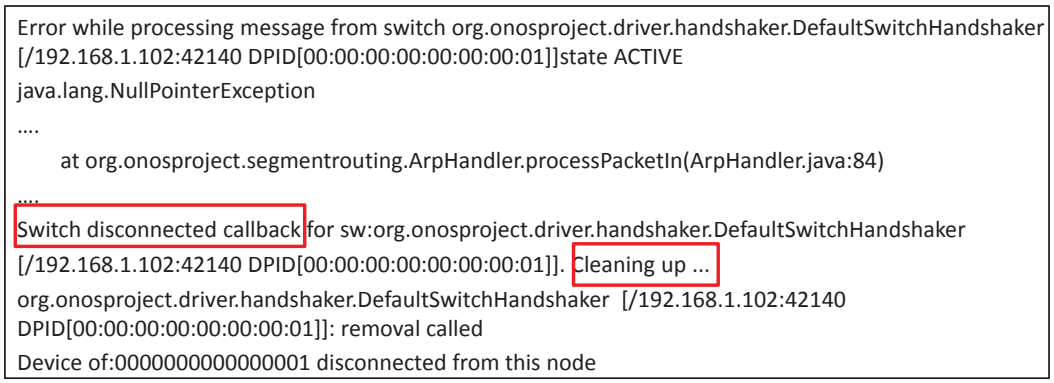


Figure 3.12: A harmful race condition in ONOS causes disconnection of a switch.

scheduling (as shown in Figure 3.10) and found that the main thread of Floodlight controller was unexpectedly terminated.

Impact #2: Switch Connection Disruption. We found 7 bugs (**Bug-5**, **Bug-6**, **Bug-7**, **Bug-8**, **Bug-9**, **Bug-11** and **Bug-12**) that could cause the SDN controller to actively close the connection to an online switch. Figure 3.11 and Figure 3.12 show stack traces reproducing this issue in Floodlight and ONOS controllers. The connection disruption is a serious issue in SDN domain since: (1) by default, the victim switch may downgrade to traditional Non-OpenFlow enabled switch and then traffic can go through it without controller's inspection; (2) an SDN controller may send instructions to clear the flow table of the victim switch when the controller recognizes a connection attempt from the switch[6]. As a result, security-related rules may also be purged.

Impact #3: Service Disruption. We also found several bugs that could interrupt the enforcement of services inside the SDN control plane, which may lead to serious logic bugs that hazard the whole SDN network.

In Floodlight, we found 3 bugs (**Bug-1**, **Bug-2**, and **Bug-3**) in the LinkDiscoveryManager application that can violate the operation of link discovery procedure. Moreover, we found 1 bug (**Bug-10**) in the Statistics application that disrupts the processing of REST requests. In addition, we located 5 such bugs in the `OFP_PACKET_IN` handler of LoadBalancer application. **Bug-5**

[6]This is an optional feature specified in OpenFlow protocol to prevent residual flow rule problem. However, we find that this feature could be enabled in most of SDN controllers.

and **Bug-6** could cause a logic flaw that leaks the physical IP address of the public server's replica. **Bug-7**, **Bug-8** and **Bug-9** could disrupt the handling of `OFP_PACKET_IN` events.

In ONOS, we found two such bugs (**Bug-11** and **Bug-12**). The bug **Bug-11** is in the SegmentRouting application that can disable the proxy ARP service and lead to the temporary block of end-to-end communication on a specific host. Similarly, the bug **Bug-12** is in the DHCPRelay application that will disable the DHCP relay service to send out DHCP reply to its clients.

In OpenDaylight, we found two such bugs. One (**Bug-13**) is in the HostTracker application, which could deny the REST API requests for creating a static host for a known host. The other (**Bug-15**) could affect the functionality of a Web UI application.

Impact #4: Service Chain Interference. We found several bugs that could violate the network visibility among various applications and could block applications from receiving their subscribed network events. In Floodlight, we found 5 such bugs (**Bug-5**, **Bug-6**, **Bug-7**, **Bug-8** and **Bug-9**) in the LoadBalancer application that could break the service chain for `OFP_PACKET_IN` event handlers. Similarly, we found 1 bug (**Bug-14**) in OpenDaylight, i.e., a concurrent `HOST_LEAVE` event can break the host event handling chain.

3.6 Related Work

TOCTTOU vulnerabilities and attacks. One infamous category of concurrency vulnerabilities is TOCTTOU (Time of Check to Time of Use) vulnerabilities widely identified in file systems, which allow attackers to violate access control checks due to non-atomicity between the check and the use on the system resources [71, 72, 73]. In this work, we study harmful race conditions in SDN networks, i.e., harmful race conditions upon shared network state variables triggered by external network events. In contrast to TOCTTOU vulnerabilities, a harmful race condition detected in this work is a more general type of concurrency errors which does not necessarily include a check operation upon race state variables.

Race Detectors. To date, researchers have developed numerous race detectors for general thread-based programs [74, 59, 58] and domain-specific programs in web and Android [75, 60, 61, 76]. However, these existing detectors do not work well for harmful race conditions discussed

in this work because (1) harmful race condition vulnerabilities are not necessary data races as discussed earlier (in many cases they are not), (2) these detectors lack SDN concurrency semantics.

In the SDN domain, SDNRacer [57, 56] proposes to detect concurrency violations in the *data plane* of SDN networks while treating the SDN control plane as a blackbox. SDNRacer utilizes happens-before relations to model SDN data plane and commutative specification to locate data plane commutative violations. Attendre [77] extends OpenFlow protocol to mitigate three kinds of data plane race conditions to facilitate packet forwarding and model checking. However, SDNRacer and Attendre are exclusively effective in the SDN data plane and fail to solve concurrency flaws in the SDN control plane, which has different semantics. In this sense, our work is complementary to those work in effectively locating unknown concurrency flaws in the SDN control plane.

Active Testing Techniques. Our active scheduling technique is inspired by the schools of active testing techniques for software testing [78, 79], which actively control thread schedules to expose certain concurrency bugs such as data races and deadlocks. Differently, our technique is specialized for the SDN controllers.

Verification and Debugging Research in SDN. Anteater [80] presents a static analysis approach to debug SDN data plane by translating network invariant verification to the boolean satisfiability problem. NICE [38] complements model checking with symbolic execution to locate operation bugs inside SDN controller applications. Vericon [81] develops a system to verify if an SDN program is correct to user-specified admissible network topologies and desired network-wide invariants. OFRewind [82] proposes to reproduce SDN operation errors by utilizing record-and-replay technique. SOFT [39] complements symbolic execution with cross checking to test interoperability of SDN switches. STS [83] leverages delta debugging algorithm to derive minimal causal sequence for SDN controller operation bugs, which can facilitate network troubleshooting and root-cause analysis. Veriflow [36] proposes a shim layer between the SDN controller and switches to check network invariants. NetPlumber [37] introduces Header Space Analysis to verify network-wide invariant at real-time. None of the above verification tools are designed to precisely

pinpoint concurrency flaws inside SDN control plane, which is the focus of this work.

Security Research in SDN. Recently, there are many studies investigating security issues in SDNs. Ropke and Holz propose that attackers can utilize rootkit techniques to subvert SDN controllers [84]. DELTA [85] presents a fuzzing-based penetration testing framework to find unknown attacks in SDN controllers. TopoGuard [28] pinpoints two new attack vectors against SDN control plane that can poison network visibility and mislead further network operation, as well as proposes mitigation approaches to fortify SDN control plane. In contrast to existing threats, in this work we study a new threat to the SDN, i.e., harmful race conditions in the SDN control plane.

To fortify SDN networks, AvantGuard [46] and FloodGuard [86] propose schemes to defend against unique Denial-of-Service attacks inside SDN networks. FortNOX [43] and SE-FloodLight [87] propose several security extensions to prevent malicious applications from violating security policies enforced in the data plane. SPHINX [47] presents a novel model representation, called flow-graph, to detect several network attacks against SDN networks. Rosemary [88] and [89] propose sandbox strategies to protect SDN control plane from malicious applications. Although some of those work could isolate some impacts introduced by the harmful race conditions, such as system crash, they are not designed to detect those concurrency flaws as we have illustrated in this work.

3.7 Limitations and Discussion

Testing Coverage. As a common drawback of dynamic analysis techniques [90], the race detection part of cannot cover all execution paths. Thus, may not cover all harmful race conditions due to its dynamic nature. Instead, it focuses on locating the vulnerabilities more accurately given an execution trace. Also, our SDN-specific input generator is designed to cover essential and remote-attacker-accessible SDN events as much as possible to pinpoint concurrency vulnerabilities in the SDN control plane. To increase the code coverage, in our future work, we plan to complement with other coverage-based techniques such as symbolic execution [91, 92].

Supporting More Controllers and Other Event-driven Systems. The current implementations of are targeting Java-based mainstream SDN controllers such as Floodlight, ONOS and

Opendaylight, which are widely adopted in both academia and industry. In fact, our technical principles and approaches are generic because the design of is based on the abstracted semantics of the SDN control plane. In that sense, we can easily port to other SDN controllers. We consider this work as a starting point for the security research on the concurrency issues inside the SDN control plane. In the future, we plan to extend our platform to other SDN controllers.

In addition to the SDN control plane and its applications, we note that harmful race conditions may occur in other multi-threaded event-driven systems, such as Web and Android applications. At high level, our approach is generic to those systems because our basic principle is to locate harmful race conditions from commutative races. In order to adapt our approach to other systems, one needs to feed with precise domain-specific models (like happens-before rules discussed in Section 3.3.1) and proper design of *Active Scheduling*.

Misuses of SDN Control Plane Northbound Interfaces (NBIs). An application may provide service functions to other applications for referencing its managed state (e.g., Switch Manager application provides switch state by the service function *getSwitch()*). If the state variable is subject to race state operations, an SDN application may misuse service functions (which are also known as NBIs) to reference network state variables from other applications. In this work, we have studied the concurrency violations introduced by specific misuses of those NBIs. However, verification and sanitization of more generalized uses of SDN control plane NBIs are still challenging issues. We plan to study these problems in future work.

3.8 Conclusion of This Work

In this work, we conduct systematic study on the concurrent programming model in SDN. From the study, We present a new attack on SDN networks that leverages harmful race conditions in the SDN control plane to crash SDN controllers, disrupt core services, steal privacy information, etc. We develop a dynamic framework including a set of novel techniques for detecting and validating harmful race conditions. Our tool has found 15 previously unknown vulnerabilities in three mainstream SDN controllers. This work contributes to enhance the security of network programmability in SDN.

4. SYSFLOW: PROVIDING SYSTEM SECURITY VISIBILITY AND PROGRAMMABILITY IN SDN

4.1 Introduction

With holistic network visibility and flexible programmability, security participators tend to use SDN to implement and deploy innovative security applications. Not only in academic environments, but also in real-world production networks, SDN, particularly its popular realization OpenFlow, has been increasingly employed with many security application scenarios ranging from network access control to network intrusion detection and network moving target defenses.

However, to archive stealthy and elusiveness, more and more emerging cyber attack campaigns tend to leverage "low-and-slow" multi-stage attack patterns, which only leave small footprints on each affected system in a short interval. Unfortunately, in many cases, existing SDN techniques fall short in detecting and preventing such advanced attacks due to it lacks system-level security visibility and programmability. For example, an SDN-based firewall can hardly prevent outgoing traffic associated with data ex-filtration attack if the leaked sensitive data is encrypted by the attacker. To this end, this work presents the design and implementation details of our project called SYSFLOW that provides system security visibility and programmability to SDN architecture. The goal of this work is to allow users to write security applications to detect, prevent and response cyber attacks in an enterprise/cloud infrastructure with unified security visibility and programmability in both network-level and system-level. To achieve the goal, we face the following challenges:

- First, can we provide a unified abstraction to model system activities and security capabilities tightly coupled with low-level details, e.g., operation systems and hardware?
- Second, with the unified abstraction, can we provide an SDN-compatible framework to effectively and efficiently enforce holistic visibility and flexible programmability?

To address the first challenge, we introduce a general system activity abstraction over existing system security capabilities. In particular, we introduce a flow-based model, namely *system flow*,

to abstract system activities. A system flow consists of 3 tuples (source, destination, and operation) to generically and formally reason about the state of diverse system activities. Moreover, based on the system flow model, we introduce *system flow rules* that can be used to represent system security intents. The key insight of a system flow rule is to complement expressive defensive actions to system flows, which can embrace a "Match-Action" paradigm. For example, a system flow rule can be used to enforce a cyber deception intent by specifying a system flow to *match* open attempts from suspicious processes to a sensitive file and conduct redirect *action* to steer the matched system flow to a decoy file for further deception/analysis.

To address the second challenge, upon the flow-based system security abstraction, we propose a novel framework, namely SYSFLOW, to enable system security visibility and programmability by enforcing flow-level security control of host system activities at run-time.

In order to integrate with SDN, SYSFLOW also embraces a two-layer architecture, including two major parts, SYSFLOW Data Plane, and SYSFLOW Controller. At the low level, the SYSFLOW Data Plane automatically enforces system flow rules to enable fine-grained responsive security actions, and dynamically update security intents (in the form of system flow rules) according to the change of contexts. At the high level, the logically centralized SYSFLOW Controller acquires a holistic view of security contexts from the low-level abstraction of host systems and provides a unified programming abstraction to facilitate the flexible implementation and deployment of diverse SYSFLOW security applications based on system flows, even across the entire infrastructure. Based on SYSFLOW, security administrators can easily extend the functions of SDN with many novel types of network/system security applications, such as file reflector, cross-host context-aware access control, programmable micro-segmentation, and cross-layer data leakage defense, on top of SYSFLOW Controller.

The key contributions of this work are summarized as follows:

- We introduce a unified programming abstraction with a novel data plane model for host systems, namely *system flow*, which can facilitate the specification and enforcement of system security intents.

- We design and implement SYSFLOW, a framework to enable SDN controller to enforce unified security intents with system security visibility and programmability.
- Our extensive evaluations show that SYSFLOW is useful to enable SDN to write various types of known/new system security applications with a minor performance overhead.

4.2 Problem Statement

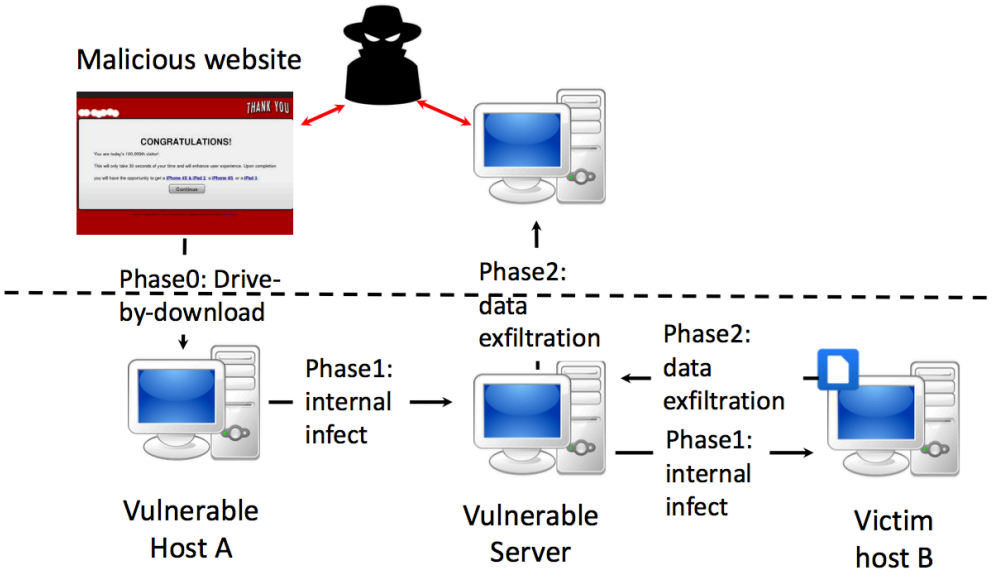


Figure 4.1: A stepping-stone data exfiltration attack.

4.2.1 Motivating Example

Figure 4.1 exhibits an abstracted multiple-stage APT attack as reported by TrapX [93]. At stage 0, the attacker lures $host_A$ to install and run a malicious executable by using a drive-by-download attack. At stage 1, the infected $host_A$ contacts and compromises another vulnerable server that has the privilege to upload data to the remote server (outside the internal network). At stage 2, the infected $host_B$ can launch a data ex-filtration attack via the vulnerable server in a stepping stone way. In the example, a security system can hardly decide which outgoing session is associated with data leakage if it only focuses on the view of the vulnerable server. The reason

Notations: Integers n, Wildcard *

Match	*ma*	::= <*src, dst, type*>
Source ID	*src*	::= *id* | *
Destination ID	*dst*	::= *id* | *
Operation Type	*type*	::= *n*
ID	*id*	::= $\{n_1, n_2, ..., n_k\}$
Action	act	::= pa | (act | act) | (act >> act)
Primitive Actions	*pa*	::= allow | deny | report| message | log | encode(tag) | decode(tag) | redirect (dst_1, dst_2) | isolation(pid_1, pid_2) | quarantine (pid_1, rid_2) | external(*id*)
Priorities	*pri*	::= *n*
System Flow Rule	*rule*	::= <*ma, act, pri, md*>

Figure 4.2: Syntax of system flow rule.

lies in that the vulnerable server will legitimately upload data to remote cloud service for archive purpose. In the example, an SDN-based security application may fail to prevent such data leakage since it can hardly decide which outgoing session is associated with data leakage if it only focuses on the view of the network traffic. The reason lies in that the vulnerable server will legitimately upload data to remote cloud service for archive purpose.

4.2.2 System Security Abstraction

In this work, we adopt a general concept of *system events*, which are regarded as interactions between programs/processes and resources. Inspired by programmable networks [1], SYSFLOW introduces *system flow rules* to model system security capabilities upon a sequence of system events.

The system flow rule is formally defined by the syntax in Figure 4.2. System flow rules are used to capture system security intents, which include *match*, *action*, *priority*.

Match. A *match* is a predicate to match a sequence of system events that have the same attributes, i.e., *source*, *destination* or *type*. The source of a system flow is an identifier for the initiator of the flow, which is an identifier of a system application. The destination of a system flow is an identifier for the receiver of the flow, which is an identifier of system resources, such as files, memories, etc. The type of system flow is used to classify the different interactions between system applications and resources, e.g., writing a file. Note that, a system flow can be used to represent an exact system event or a group of system events with the same pattern by using a wildcard notation (*). For example, a system flow can be specified as $\{src : *,\ dst : file_1,\ type : file\_op\_write\}$ to match system events representing any process writes to $file_1$.

Action. A system flow rule uses a list of *primitive actions* to specify how the system events should be processed. The intuitive primitive actions are *allow* and *deny*, which are used to enforce explicit access control upon matched system events. Moreover, *isolation* is used to isolate communications (e.g., IPC) between two different processes. *quarantine* aims to constraint a specific process under a fine-grained resource context. Also, *log* can be used to record system events for further analysis, e.g., digital forensics. *encode* is to push contextual tag into outgoing network packets, and *decode* is to check the contextual tag from incoming network packets, which can be used to enforce cross-host information flow tracking. Furthermore, *redirect* aims to change the system flow to a new location, which is helpful to enforce a deception-based defense. Besides, *alert*

Table 4.1: Security actions defined in SYSFLOW.

Name	**Descriptions**
Allow	permit matched system events pass through
Deny	prevent matched system events pass through
Encode	push contextual tag into outgoing packets
Decode	check contextual tag from incoming packets
Redirect	redirect matched system events to a new destination
Isolation	isolate a process from other processes
Quarantine	restrict a process under a specific running context
Alert	send alerts to administrators
Log	log matched events based on conditions
Message	pop up a prompting window to notify the host user.

and *message* are proposed to notify administrators or host users of suspicious/abnormal events. We note that the defined security actions are basic security primitives that we abstract from many existing system security applications. In addition, a system flow can also use *external* primitive action that is customized by a security application developer.

Priority. An integer-based priority is used to disambiguate rules with overlapping patterns. If a system event matches multiple system flow rules, only the highest-priority rule is applied. In this case, the priority can be used to explicitly specify the matching order to resolve potential conflicts from a set of system flow rules.

System Flow Rule Example. The system flow rule can be used to enforce a wide variety of basic security intents. For example, a system flow rule "*src: proc_2, dst: mem_1, type: mem_allocation, priority:0, actions: report*" can be used to report to the controller about memory allocation events from a specific process to a specific memory location for further analysis. Another system flow rule "*src: *, dst: file_1, type: file_open, priority:0, actions: block*" can be used to disallow any open operations on a specific file.

4.2.3 System Overview

As illustrated in Figure 4.3, SYSFLOW embraces a two-layered programmable design that includes two major parts, SYSFLOW *Data Plane* and SYSFLOW *Controller* .

The SYSFLOW Data Plane runs in a target host system. The SYSFLOW Data Plane Daemon resides in the user space of the system that is used to intercept communications between the SYSFLOW Controller and the Flow Table Manager in the kernel. It talks with the SYSFLOW Controller by using the SYSFLOW control messages and interacts with the Flow Table Manager to manipulate flow tables accordingly. The Event Generator abstracts low-level system activities to generate system events and further inputs those system events to the Flow Table Manager. The Flow Table Manager maintains system flow rules (in a flow table) to match system events and trigger the Action Scheduler to enforce corresponding actions to control system activities.

The SYSFLOW Controller works as a logically centralized control and management nexus that provides interfaces to SDN controller to collect context information from all host systems running

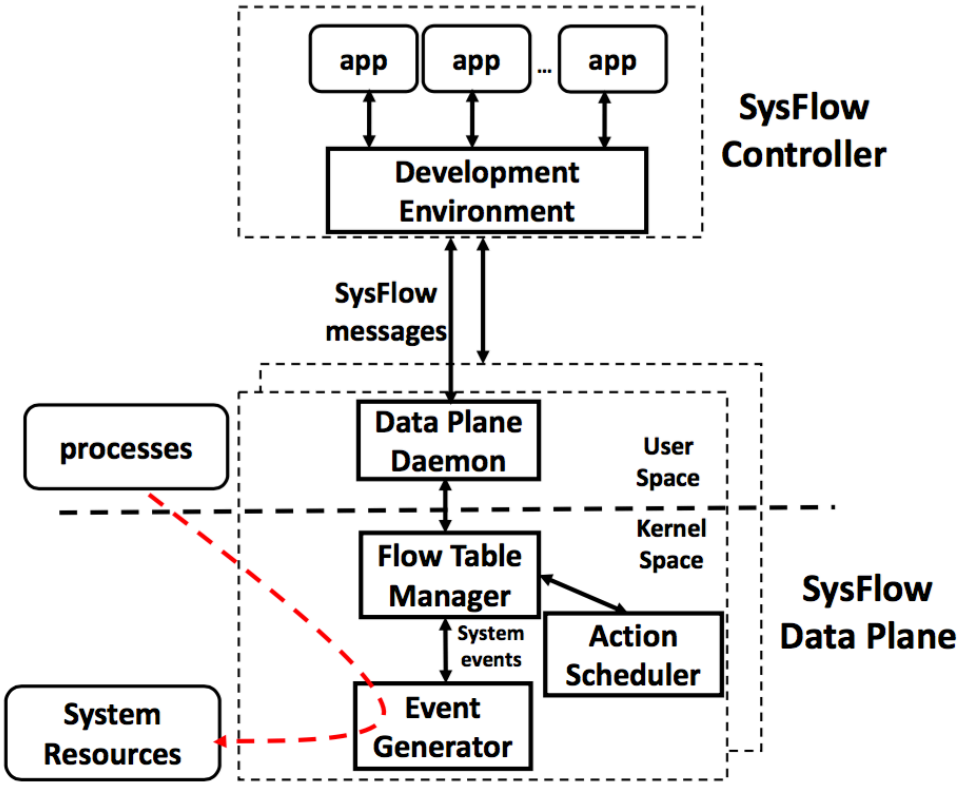


Figure 4.3: SYSFLOW components.

the SYSFLOW Data Plane and install system flow rules accordingly. The controller provides a unified high-level programming abstraction to facilitate SDN controller with system security visibility and programmability.

Figure 4.4 illustrates a typical workflow of SYSFLOW. The security applications can update system flow rules into the SYSFLOW Data Plane via control channels (e.g., SYSFLOW Controller and SYSFLOW Data Plane Daemon). The system flow rules are further used by the Flow Table Manager to match system events abstracted from low-level system activities. If a system event matches any system flow rule, the Action Scheduler will execute security actions specified in matched flow rules, which conduct a security control upon system activities. Also, some matched system flow rules will issue flow reports to send context information to security applications (as

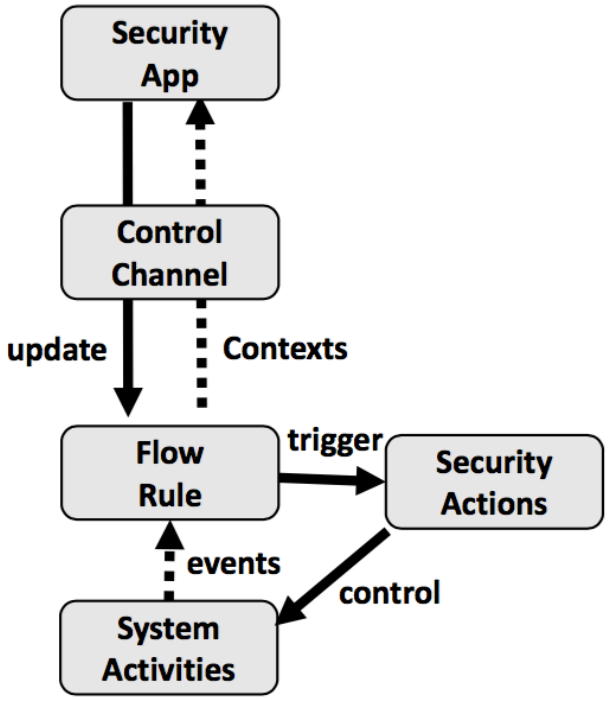


Figure 4.4: SYSFLOW workflow.

their requests). Then, those security applications may install responsive flow rules in the SYSFLOW Data Plane.

4.2.4 Threat Model

Similar to prior system security approaches [94, 95, 96, 97, 98], we first assume that the kernel, in which the SYSFLOW Data Plane is running, the communication channels, and the server running the SYSFLOW Controller are trusted computing base (TCB). We consider that an adversary may attempt to compromise the availability or privacy of the system resources protected by SYSFLOW. In the case, the adversary (in the user space), for instance, may install malware/ransomware, exploit running processes, or launch denial-of-service (DoS) attacks.

In addition, we make the following assumptions. First, attacks will happen only after the initiation of the SYSFLOW Data Plane and the SYSFLOW Controller. Second, attacks based on

hardware Trojans/Rootkits and side/covert channels of shared system resources are out of the scope of the work. Third, SYSFLOW can leverage state-of-the-art integrity-checking mechanisms [99, 100, 101, 102, 103] to determine if there are any compromises against SYSFLOW components, especially the SYSFLOW Data Plane Daemon in the user space.

4.3 SYSFLOW Data Plane Design

In this section, we detail our design for the SYSFLOW Data Plane. First, the SYSFLOW Data Plane abstracts system details to enable unified system security programmability. Also, the SYSFLOW Data Plane leverages an efficient system flow rule management scheme to facilitate dynamic security intent update at run-time.

Generating System Events. The Event Generator component in SYSFLOW Data Plane is used to monitor and control system-level activities (e.g., system calls) in the OS kernel by bridging the semantic gaps between different implementations (e.g., types and versions) of operating systems. Some examples of system events are listed in Table 4.3. In particular, Event Generator consists of hooks for critical operations to generated various system events. The Event Generator further inputs system events to Flow Table Manager to reference system flow rules in the flow table and enforce corresponding security actions.

Resolving Resource Identifier. In many cases, the security application developers encounter semantic gaps for system objects, i.e., they can hardly specify the identifier of system objects. For example, a security application developer may not know the identifiers (i.e., *UUID* and *inode number*) of tax files in the file system of the victim host when they want to write security apps to prevent the ex-filtration attacks. Instead, they may be aware of the file name and possibly the path. In order to bridge the semantic gap, Flow Table Manager enables an identifier binding and resolution service for system objects.

At run-time, the Flow Table Manager maintains the profile table for system objects (as shown in Table 4.2). For example, Flow Table Manager will keep the executable program name of a process in addition to the process identifier. When receiving the requests to update flow rules that include attributes of system objects instead of the identifiers, the Flow Table Manager will refer to

Table 4.2: Example system object attributes.

System Object	Example Profile
Process	pid, ppid, pname, stime, etime, etc
File	uuid, inode number, fname, path, etc
Socket	uuid, inode number, local/remote addr/port, etc

the profile table to retrieve the identifier of the system object. We note that, in some cases, the map from the attributes of a system object to its identifier is not unique. For example, the name of an executable program may map to multiple running processes. In the case, we will install flow rules for each of them. Besides, the binding service will also monitor the change of the mapping from non-identifier profiles to the identifier (e.g., from name to *UUID* and *inode number* for a file) at run-time and update the inductive flow rule accordingly.

4.3.1 Efficient Flow Rule Management

The low-level security intents of SYSFLOW are embedded in a table of system flow rules, called SYSFLOW flow table. We note that the flow rule in SYSFLOW flow table can have two types of match patterns, i.e., *exact match* and *wildcard match*. In order to support both of them, an intuitive solution for SYSFLOW flow table is to use a bit-wise classification, i.e., we compare the incoming system events with all system flow rules in the table bit by bit. However, in such a solution, the time complexity for flow table lookups and updates is $O(R)$, in which R is the number of system flow rules resided in a running system. As a result, security intent enforcement will incur a high latency in the data plane.

In order to support an efficient flow table update and query, we adopt Tuple Space Search (TSS) classification algorithm [104]. The key insight of TSS classification is to realize a flow table as a set of hash tables (here, the hash table is referred to as tuple). In order to classify each flow table, we use a 3-bit mask to specify the range of each tuple. The value of "1" in the bit of the mask means the matching field of the tuple and the value of "0" denotes ignoring (wildcard) field. For example, "111" means the flow rule needs all exact match for the system event and "110" means the flow rule only matches source and destination objects (and ignores the operation code). In

each hash table, it stores the hashed key for each specific system flow rule with the same mask. Based on the TSS algorithm, Flow Table Manager can provide an efficient flow table management, e.g., flow rule update/lookup, with time complexity of $O(1)$. [1] We detail the system flow table management algorithms as follows.

Flow Table Update. As shown in Algorithm 1, the Flow Table Manager first computes the hash value from the flow pattern and its mask value (line 2). If the SYSFLOW Controller requests to insert a new flow rule in the flow table, the manager will locate a specific hash table associated with the mask value of the flow rule (line 3). If no match hash table is found, it will add a new hash table into the flow table (lines 4-5). Alternatively, it will further search if there is a duplicated flow pattern in the matched flow table (line 7). If duplication is found, it will update the existing flow rule in the flow table (lines 12-13) instead of inserting a new one (line 9). Otherwise, if the controller instructs to remove a flow rule, the manager will delete corresponding elements in the hash table (line 17).

Algorithm 1 SYSFLOW Flow Table Update

Require: FR : the SYSFLOW flow rule, M : the mask of the flow rule, C : the command of the flow rule, FT: flow table in the data plane

```
1:  hash = compute_masked_hash(FR.match, M)
2:  if C == FlowRuleAdd then
3:      hash_table = T.hash_tables.get(M)
4:      if hash_table == NULL then
5:          T.hash_tables.put(M, hash_table)
6:      end if
7:      if hash_table.contains(hash) == false then
8:          /*insert a new flow rule*/
9:          hash_table.add(hash, FR)
10:     else
11:         /*update the existing flow rule*/
12:         hash_table.remove(hash)
13:         hash_table.add(hash, FR)
14:     end if
15: else
16:     /*remove an existing flow rule*/
17:     hash_table.remove(hash)
18: end if
19: return FT
```

[1] The flow rule lookup with TSS needs T hashed memory accesses, where T is a constant value (i.e., 3 in this work) of the number of tuples. The flow rule update with TSS needs 1 hashed memory accesses.

Algorithm 2 SYSFLOW Flow Table Lookup

Require: E : the system event, FT: flow table in the data plane
Ensure: FRs: the matched flow rules with the highest priority for the system event

```
 1: FRs = { }
 2: max_priority = MIN_VALUE;
 3: for (mask,hash_table) in T.hash_tables do
 4:     /*detect all matched flow rules based on the system event*/
 5:     match = extract_flow_match(E)
 6:     hash = compute_masked_hash(match, mask)
 7:     if hash_table.contains(hash) == true then
 8:         flow_rule = hash_table.get(hash)
 9:         if flow_rule.priority > max_priority then
10:             max_priority = flow_rule.priority
11:             /*remove all located flow rules with a lower priority*/
12:             FRs.removeall()
13:             FRs.add(flow_rule)
14:         else if max_priority == flow_rule.priority then
15:             FRs.add(flow_rule)
16:         end if
17:     end if
18: end for
19: return FRs
```

Flow Table Query. A flow table lookup algorithm is provided in Algorithm 2. When a system event arrives, the SYSFLOW Data Plane will iterate all hash tables in the SYSFLOW flow table to find matching flow rules: It first extracts the flow pattern from the system event (line 3) and computes the hash value based on the extracted pattern and the mask value of the hash table (line 4). Then, the manager searches the hash value from the flow table. Once finding a matching rule, it will check the priority of the flow rule with previously added one and store those rules with the highest priority (lines 9-15). If Flow Table Manager cannot find any matching flow rule, the manager will return an empty flow rule set and enforce a default action (i.e., deny or accept) which is pre-configured by the administrator. Note that, the flow table query procedure does not resolve the conflicts at run-time, i.e., it assumes that there are no conflicting actions from multiple detected flow rules based on a specific system event.

4.4 SYSFLOW Controller Design

The SYSFLOW Controller provides a high-level programming framework to facilitate the SDN controllers with system visibility and programmability based on the model of system flow rules. In this section, we will first describe the run-time development environment provided by SYSFLOW Controller that can help security developers to compose different system security intents. Next, we discuss how SYSFLOW Controller can provide global visibility of system contexts for security applications.

4.4.1 SYSFLOW Development Environment

High-level Data Types. Instead of reasoning about the low-level identifiers of system objects, we provide a higher-level abstraction to facilitate security application developers to enforce their security intents. For example, the developers can specify a file with its attributes e.g., name and path for a file. Then, SYSFLOW Data Plane will leverage Identifier Binding and Resolution service to resolve the identifier of system objects at run-time.

SYSFLOW **Controller APIs.** The SYSFLOW Controller provides several system-flow-related SYSFLOW APIs, e.g., *installSysFlowRule*, *revokeSysFlowRule*, and *handleSysFlowReport*, to enable the development of security applications. The *installSysFlowRule* is to instruct the SYSFLOW controller to install a system flow rule into a host system running SYSFLOW Data Plane. The *revokeSysFlowRule* is to instruct the SYSFLOW controller to revoke an installed system flow rule in a host system running SYSFLOW Data Plane. The *handleSysFlowReport* allows SYSFLOW applications to process system flow rule reports to extract contexts of host systems. Moreover, the SYSFLOW Controller also provides other callback functions, e.g., timer-based callbacks or object-change callbacks, to ease the development of SYSFLOW applications.

4.4.2 Global Visibility

By design, SYSFLOW Controller also envisions security contexts (collected from SYSFLOW Data Plane) to enforce system security intents among multiple systems in an infrastructure. A security application can collect system contexts by registering flow report handler functions in

SYSFLOW Controller to process flow status reports from installed monitoring flow rules in a target system. We discuss several types of security contexts supported in SYSFLOW as follows. We consider security contexts from system events, which abstract states, attributes, and interactions between processes and system resources in a host system. For example, a process connected to a socket (whose remote IP address is associated in a blacklist) can be used as a security context to allow security application to block such a process to visit user-specified sensitive files.

Cross-Host Context Sharing. Furthermore, SYSFLOW enables cross-host visibility to allow dynamic re-configurations of security intents. In particular, SYSFLOW supports two types of cross-host context sharing, i.e., *reactive controller updating* and *proactive packet tagging*. In a reactive updating manner, a security application leverage SYSFLOW Controller to install multiple monitoring flow rules in various systems, register flow reports to acquire system contexts from different host systems, and optionally update response flow rules based on monitored contexts. In a proactive packet tagging manner, a security application can proactively install flow rules to leverage *encode* and *decode* security actions to encapsulate system-level contexts to the tags of outgoing packets and enforce different security actions based on tags.

4.5 SYSFLOW Implementation

A prototype of SYSFLOW Controller has been implemented in Java [2] and built upon Java Non-Blocking IO (NIO) API to achieve high event processing throughput. Currently, SYSFLOW security apps can be developed in Java as well and instantiated as a module of the SYSFLOW Controller. We have implemented a prototype of SYSFLOW Data Plane on top of Linux in C. SYSFLOW captures a list of system events based on Linux Security Modules (LSM) framework [105].

Table 4.3 lists some hooks in the Linux kernel used to generate system events in the SYSFLOW kernel plugin module. To enable file permission access control, SYSFLOW can generate three different system events, corresponding to opening, reading, and writing a file, respectively. The *file_ioctl* event is designed to allow SYSFLOW capture manipulations of the underlying device parameters. The system events relevant to inode operations provide SYSFLOW with the visibility of

[2] We note that the implementation of SYSFLOW Controller is programming language agnostic.

Table 4.3: Linux kernel hooks for system events

LSM hook	System event
inode_permission	file_open
file_permission	file_read, file_write
file_ioctl	file_ioctl
dentry_open	dentry_open
inode_create	inode_create
inode_mknod	dev_inode_create
inode_symlink	symlink_create
inode_mkdir	dir_create
inode_link	file_create
inode_unlink	file_remove
inode_rmdir	dir_remove
socket_recvmsg socket_sock_rcv_skb	socket_read
socket_sendmsg	socket_write

any creation, removal, and modification of the underlying file system data structures. The file_open event can be generated through the *inode_permission* hook. The file_read and file_write events can be generated through the *file_permission* hook. The socket system events are implemented to enable fine-grained control over various socket behaviors. It provides more than ten hooks to control socket behaviors. We will extend the supported system events over time.

We next discuss the implementation of built-in security actions in SYSFLOW (listed in Table 4.1). The *allow* and *deny* directives allow and deny, respectively, certain operations, including *open*, *read*, and *write* a file. These two directives can be easily enforced through the *file_permission* or *inode_permission* hooks. The *quarantine* and *isolation* directives are used to set up a group of system flow rules with *allow* and *deny* actions. The *redirect* directive is intended for redirecting operations to another object. For example one can redirect the "open file" operation to a decoy file so that a process will open the decoy file instead of the original file. However, the enforcement of *redirect* action is quite challenging due to only relying on the existing hooks of LSM. We therefore determine to place an additional hook to handle the *redirect* action. Since a process in the user space manipulates a file through a file descriptor, we place a hook (*fd_bind*) before the file descriptor is bound to a specific inode. By invoking the *fd_bind* hook, one can enforce the *redi-*

rect action via replacing the inode of the original file with the inode of any other files according to the security intents. The *encode* directive is for tagging the outgoing packets, and we can get the tagging information from the incoming packets by *decode* directive. One challenge to implement *encode* and *decode* is to associate process identifiers with their outgoing packets, since the retrieval of packet instance in LSM hooks and the retrieval of process identifiers in NetFilter hooks are not deterministic. Hence, we embed two new hooks in LSM from Linux kernel functions (i.e., ip_queue_xmit() for *encode* and ip_rcv() for *decode*), which can help correlate process identifiers with their outgoing packets. Then, the *encode* and *decode* leverage packet tags (using IP TOS filed in outgoing or incoming packets) to exchange contexts cross multiple hosts. The *alert* directive is implemented as *flow rule status report* messages that notify security administrators. The *log* directive is implemented to write to a system log file, "/var/log/messages", in Linux, and other system log files for other operating systems.

4.6 Evaluation

In this section, we first showcase the effectiveness of SYSFLOW to enable SDN to defend against advanced cyber attacks with system-level visibility and programmability. We then present our experimental results from measuring the performance of SYSFLOW on its overhead that imposes on normal system-level operations, the efficiency of dynamic flow control, and the scalability of SYSFLOW Controller. In our experiments, we hosted the SYSFLOW Controller and the SYSFLOW Data Plane on machines running GNU/Linux Ubuntu 12.04 with dual-core CPU and 8 GB memory.

4.6.1 Use Case: Cross-Layer Data Leakage Prevention

In addition, we show that how SYSFLOW can benefit SDN (as illustrated in Figure 4.5) to write hybrid security applications to prevent advanced cyber attacks we mentioned in motivating example. We note that such security application enforces cross-layer data leakage prevention by using both system flows and network flows. The security app installs system flow rules into the victim host and other hosts in an infrastructure (①). For the victim host, the SYSFLOW Controller

installs a system flow rule to track the system-level information flow from the sensitive file to any processes. If an access to a sensitive file is observed, a report is sent to the SYSFLOW controller (②). When the controller receives the *report* message from the victim's flow rule, it will reactively install a tracking flow rule to encode a tag for all outgoing traffic from the process that accesses the sensitive file (③). Since the SYSFLOW Controller has already installed flow rules on other hosts to instruct SYSFLOW Data Plane to report any observation of tagged packets received from the socket (④ and ⑥), when the SYSFLOW Controller receives the *report* from SYSFLOW Data Plane, it responsively installs flow rules to propagate the tag for outgoing traffic (⑤ and ⑦). Moreover, the administrator can improve the efficiency for tag propagation by placing the logic of *report* event handler into the SYSFLOW Data Plane as a SYSFLOW external security function, which can mitigate the communication overhead between the SYSFLOW Controller and the SYSFLOW Data Plane during the installation of tag-propagation flow rules. Furthermore, the app defends against data leakage attacks by using SDN controller to filter tagged traffic at the broader of the infrastructure. In this use case, we note that the system-level visibility and programmability can effectively facilitate SDN to enforce more advanced defensive solutions against emerging infrastructure-wide cyber attacks.

4.6.2 Performance Measurement on SYSFLOW Data Plane

In this section, we present our experimental results from evaluating the performance of SYSFLOW Data Plane for micro-benchmark tests, macro-benchmark tests, and scalability tests. In the following evaluations, we leverage *baseline* to refer to systems running an unmodified Linux kernel.

Micro-Benchmark results. We used LMBench [106] to evaluate the run-time performance of system calls, file operations, memory latencies, and socket I/O throughput. Table 4.4 depicts the comparison between the baseline and SYSFLOW with 1000 system flow rules. The file operations include *read*, *write*, *open/close*, *delete*, and *create*. Our evaluation results indicate that SYSFLOW mainly impacts the file operations. But for system calls, memory latencies, and socket I/O, SYSFLOW only introduces negligible overhead.

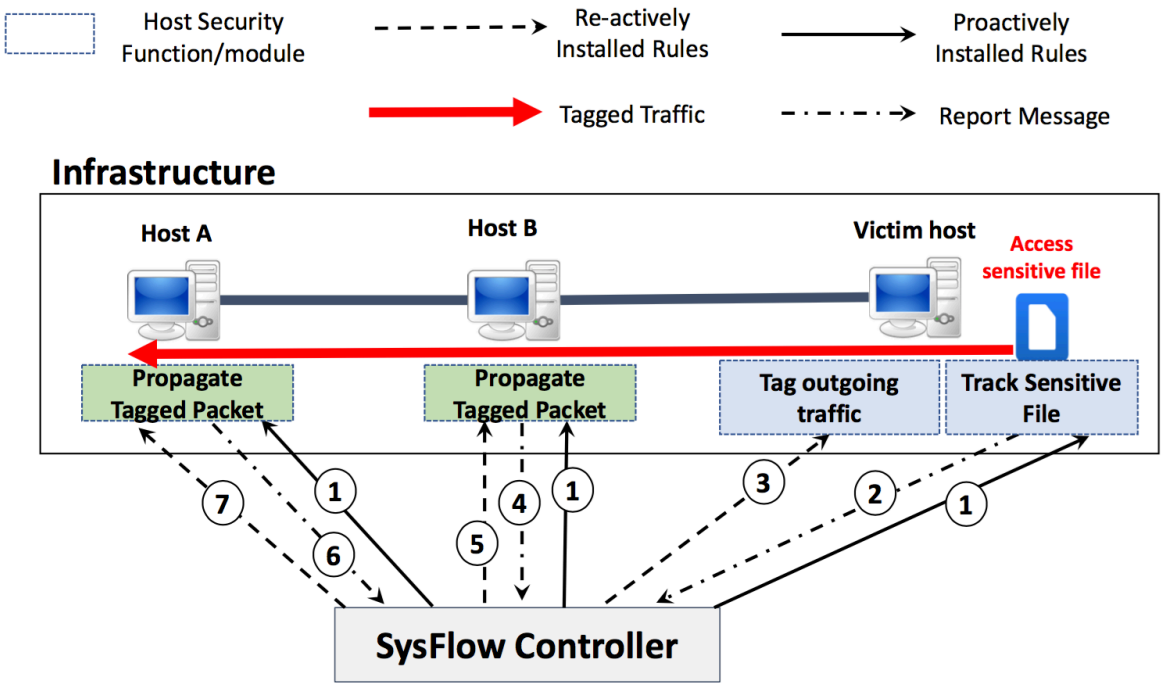


Figure 4.5: High-level idea of how infrastructure-wide data leakage investigation is realized via SYSFLOW.

Table 4.4: Micro-benchmark results for SYSFLOW Data Plane.

System Operation	Baseline	SysFlow
Latency of system operations in μs (smaller is better)		
file read	0.319	0.467 (+46.5%)
file write	1.844	2.103 (+14.1%)
file open/close	1.069	1.295 (+21.1%)
file create (0k)	6.159	6.625 (+7.6%)
file create (10k)	53.917	64.613 (+19.5%)
file delete (0k)	5.178	5.402 (+4.3%)
file delete (10k)	18.019	20.5154 (+13.9%)
syscall	0.054	0.054 (+1.4%)
mmap latency	9.803	9.826 (+0.2%)
pipe latency	6.591	6.616 (+0.4%)
Socket throughput pps (larger is better)		
socket I/O	885	883 (-0.22%)

Macro-Benchmark Results. We also tested SYSFLOW Data Plane with macro-benchmarks including web server performance, file transmission performance, and database online transaction

Table 4.5: Macro-benchmark results for SYSFLOW Data Plane.

Type	Baseline	SysFlow
Web Server (Nginx) Performance		
10K total requests with 500 concurrent connections		
Requests per second	3,533	3,502 (-1.0%)
Time per request (ms)	141	143 (+1.4%)
File Transfer (wget) Performance for 1 GB file		
Time to Complete (s)	21.8	21.9 (+0.4%)
Throughput (MB/s)	48.0	47.7 (-0.6%)
Database (MySQL) Performance with 1M records		
Transactions per second	540.6	538.6 (-0.4%)

processing performance. In all of those tests, we run the SYSFLOW Data Plane in both server side and client side with 1000 system flow rules. For the web server performance, we used a host running ApacheBench [107] to test the performance of a Nginx server by sending 10,000 requests with 500 concurrent connections. To test the file transmission performance, we used wget benchmark [108] in a host to test the transmission of a 1 GB file from a server. For the database performance, we used sysbench [109] to test a database server with 1 million records. The results (Table 4.5) show the SYSFLOW Data Plane introduces negligible overhead on the operations of real-world applications even across different hosts.

Scalability with Flow Rules Moreover, we tested the scalability of SYSFLOW Data Plane using different numbers of system flow rules. Figures 4.6, 4.7, and 4.8 depict the cumulative distribution function (CDF) of the performance of three most frequently used operations, file *read*, file *write*, and socket I/O. We increased the number of system flow rules from 1000 up to 5000 in our experiments. For file operations and socket operations, SYSFLOW can scale well. The number of system flow rules does not make a significant difference to the performance since the system flow table is implemented through a hash table. The socket I/O operation scales almost the same as the number of system flow rules increases.

In addition, we tested the memory overhead introduced by the SYSFLOW Data Plane through the *top* Linux command with different numbers of system flow rules. The result shows the memory usage is about 400 KB for 1000 flow rules and it grows linearly with the number of system flow

rules inserted. Hence, the SYSFLOW Data Plane is scalable to contain system flow rules for various system security intents.

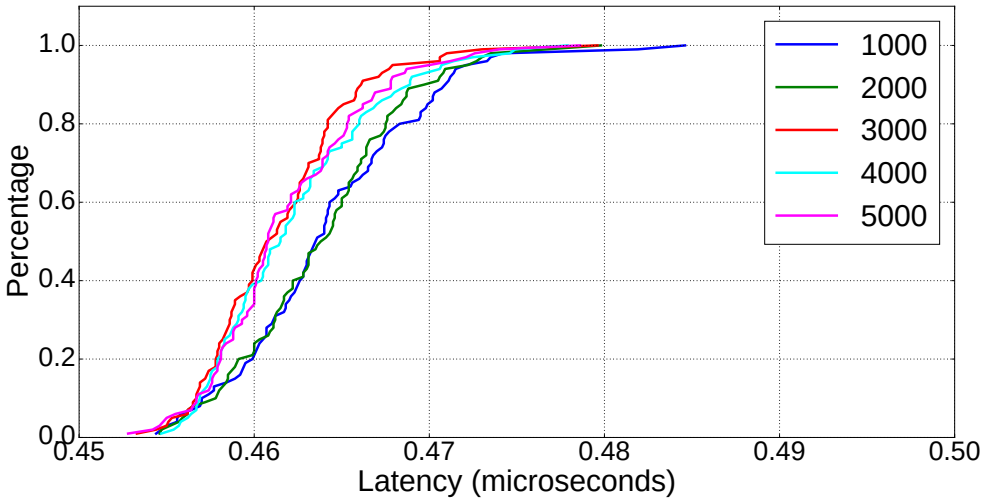


Figure 4.6: CDF of the latency of read operation with various numbers of system flow rules installed.

4.6.3 Efficiency of Dynamic Reconfiguration

Latency for flow rule update. First, we tested the latency of data plane for updating system flow rules proactively. Our controller is deployed in our university local network. We measured the flow rule update delay from the SYSFLOW controller to the flow rule table in the data plane by using a test app to send out flow modification messages. The result shows the average latency of such a proactive flow rule update is totally 10.52 milliseconds, in which the event transmission latency from the SYSFLOW Controller to the SYSFLOW Data Plane is about 9.60 milliseconds. From the result, we observed that the main factor to affect flow updates is the network latency, not the flow rule installation or matching. In this case, deploying the SYSFLOW Controller and the SYSFLOW Data Plane in a low latency network [110] may further improve the efficiency of dynamic flow control.

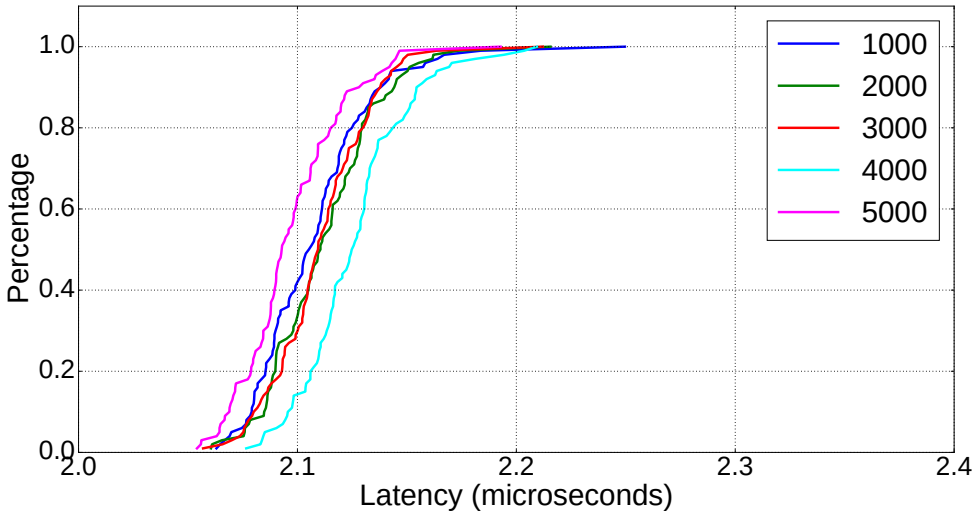


Figure 4.7: CDF of latency of write operation with various numbers of system flow rules installed.

Sensitivity of flow rule update to residual flow rules. We added a test code snippet in the SYSFLOW Data Plane Daemon that leverages the *gettimeofday* API with microsecond timestamp to measure the latency of inserting/updating/deleting 1000 flow rules cumulatively (from 1000 flow rules to 5000 flow rules). We repeated the measurement five times for each case. Figure 4.9 shows the average latencies for the Flow Table Manager to handle *flow rule modification* messages. We could observe that the Flow Table Manager can efficiently handle all types of *flow rule modification* messages, e.g., the average latency is around 2 ms for inserting 1000 flow rules when there exist 1000 flow rules. Thus, the time of inserting rules is almost negligible. We also observed that the residual flow rules do not have negative effects on the flow rule modifications since the latencies for different scenarios are in a constant trend. As a conclusion, the design and implementation of the Flow Table Manager can support a rapid reconfiguration for high-level security intents.

4.6.4 Controller Scalability

We tested the event processing throughput of a single SYSFLOW controller. Our test generator sent flow rule status report messages to the controller as fast as possible. The SYSFLOW controller (assigned with 4 GB RAM) run a stateful app that installs a new flow-mod message once receiving

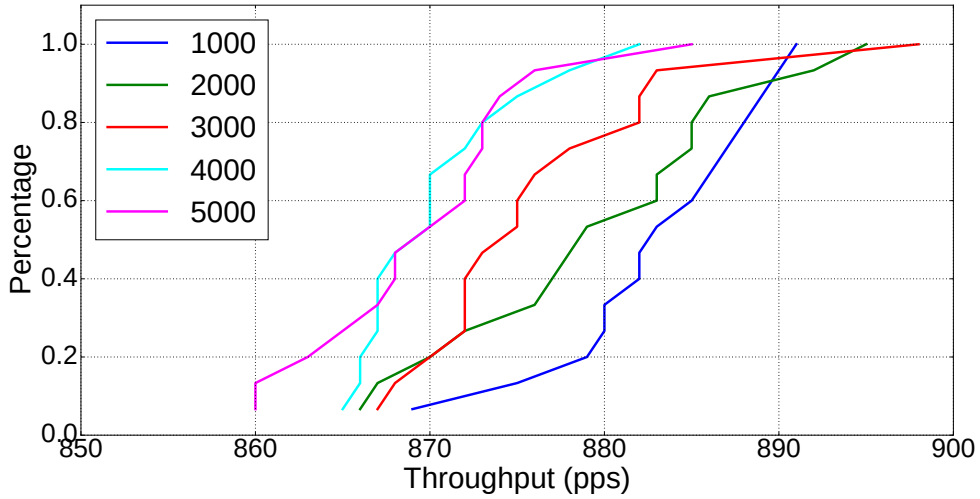


Figure 4.8: CDF of Socket I/O throughput with different numbers of system flow rules installed.

a flow report. In this case, we increased the number of generators and measured the observed output throughput of flow mod messages. The result shows the SYSFLOW controller (and its applications) can handle around 12,000 events every second. In this case, suppose that a SYSFLOW Data Plane generates 5 event per second (in the context of [111]), current implementation of a single SYSFLOW Controller (even on a low-end machine) can support around 2,400 host systems running SYSFLOW Data Plane.

4.7 Related Work

Recently, there are a couple of work propose to implement diverse security applications upon SDN due to its holistic network visibility and flexible network programmability. For example, SPHINX [47] and Veriflow [36] propose to leverage SDN techniques to detect network traffic anomalies. Also, some prior work [112, 22, 113] propose to defend against DDoS attacks by using the power of SDN. Moreover, moving target defense systems [44, 114] are proposed in SDN to delay or prevent attacks on a system. Furthermore, many work [115, 116, 117] introduce SDN-based cyber deception schemes to deploy honeypot/honeynet to lure attack traffic and investigate their attack patterns. In addition, FRESCO [11] presents an SDN-based security application development

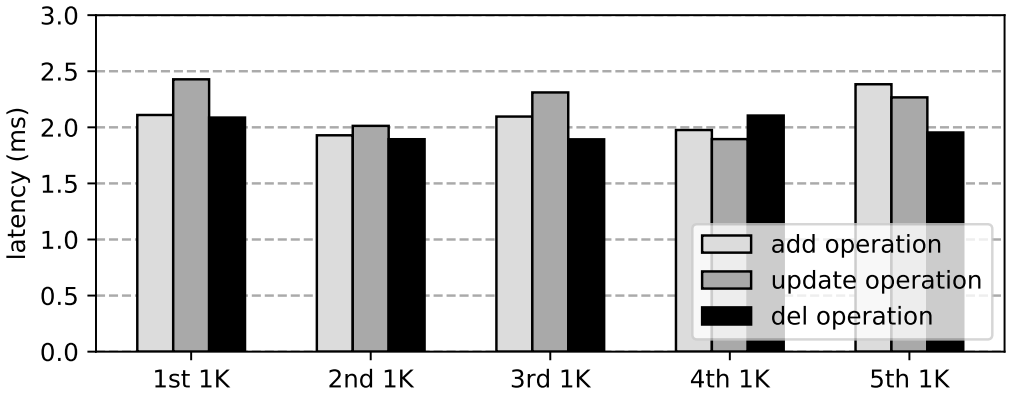


Figure 4.9: Composition of system flow rules.

framework to compose network security services by using a scripting language.

However, all of those work can only used to detect and defend network attacks since SDN only covers network-level visibility and programmability. In contrast, in this work, SYSFLOW aims to enable SDN to detect more advanced attacks (e.g., involving malicious activities inside host systems) by providing system-level visibility and programmability.

4.8 Conclusion of This Work

In this work, we have presented SYSFLOW, a novel system security development framework to facilitate SDN with flow-level security control of host system activities at run time. SYSFLOW abstracts low-level system activities into a flow-based abstraction, provides dynamic flow-level control at run time, and offers unified programmability to users to specify their security intents effectively. The evaluation results show that SYSFLOW is useful to extend SDN to develop advanced security apps with system-level visibility and programmability and only introduces acceptable runtime overhead.

5.1 Lessons Learned

As we have highlighted earlier regarding under-explored security issues on SDN innovative features (i.e., network visibility and programmability), in-depth security analysis helps us to design new security solutions. From the success of our findings, and proposed defensive insights, we have learned the following meaningful lessons:

For the security analysis on SDN infrastructure, we should pay attention to both design and implementation flaws. Our located security issues in SDN lie in two aspects, i.e., design flaws and implementation flaws. However, both of them can cause serious security and reliability issues to most SDN controllers. On the one hand, most of the existing SDN controllers and switch vendors follow a particular design convention for handling topology management. Since the first reference SDN controller implementation (i.e., NOX [118]), almost all the implementations implicitly follow its design, including OpenFlow Discovery Protocol (OFDP) and Host Tracking Service. This may be a root cause for explaining that all the controllers expose similar vulnerabilities in terms of topology management. On the other hand, harmful race conditions are rooted in implementation defects of SDN applications. However, it is extremely difficult, if not impossible, for developers to write bug-free SDN applications, especially in a concurrent programming model. Not to mention, SDN controllers embrace modular design, in which different groups of developers contributes to parts of the same function but lacks proper communications. In this sense, to secure SDN architecture, our security analysis should cover both its design and implementation aspects.

Our in-depth security analysis of SDN innovations could be an effective way to defend against SDN unique threats. In this thesis, we note that existing solutions can hardly address our located SDN security issues. The insight lies in that our located security vulnerabilities are unique to SDN innovations, i.e., holistic network visibility and flexible network programmability, which are rarely explored before (e.g., security omissions in newly introduced OFDP) and have

different natures with threats in other systems (e.g., harmful race conditions in the SDN control plane). By understanding those security threats via in-depth security analysis, we can effectively design detection solutions to locate those malicious activities (e.g., network poisoning attacks) and vulnerabilities (e.g., harmful race conditions) in SDN.

Our reported security issues and their mitigation schemes can stimulate more attentions and considerations to secure SDN architecture. In this thesis, we detect two categories of security threats against SDN architecture, i.e., topology poisoning attacks and harmful race conditions. Once we locate them, we proactively report their details and potential security advisories to victim SDN communities/vendors and Common Vulnerabilities and Exposures (CVE) authority. We received four CVE entries for our reported vulnerabilities, i.e., CVE-2015-1610, CVE-2015-1611, CVE-2015-1612, and CVE-2015-6569. We also note that our reports and paper publications [28, 29] draw many attentions from both academia and the SDN industries to put more security considerations to enhance the security of network visibility and programmability in SDN, such as [119, 120, 121, 122, 123, 124, 125]. As a result, to date, most of active-maintained SDN controllers are patched against our reported security issues. For example, the mainstream SDN controllers, i.e., Floodlight, ONOS, and OpenDaylight, have currently included our proposed HMAC-based authentication scheme to prevent fake-LLDP-based topology poisoning attacks. Also, Floodlight and ONOS controllers are correctly patched for 12 harmful race conditions as detected by our CONGUARD framework.

A unified control plane with global visibility and programmability can benefit the security of the whole infrastructure. Nowadays, more and more cyber attacks have emerged to threaten modern infrastructures, e.g., cloud or enterprise. As a novel programmable network paradigm, SDN enables many merits to defend against emerging attacks, e.g., agile network security intent programming, network-wide threat detection, and dynamic threat responses. Even with centralized control of network devices, SDN still lacks entire infrastructure-wide security capabilities, e.g., in host systems. In this thesis, we present SYSFLOW to enhance system security visibility and programmability to the SDN control plane, which showcases many innovative security benefits,

e.g., effective detection of stepping-stone data ex-filtration attacks. We consider such a successful experience can also motivate us to unify more security capabilities from other systems/components in modern infrastructures, such as virtual machines, containers, and host network stacks.

5.2 Implications on Secure SDN Architecture

Figure 5.1 exhibits a generalized secure SDN architecture by integrating and extending our proposed three components, i.e., TOPOGUARD, CONGUARD, and SYSFLOW.

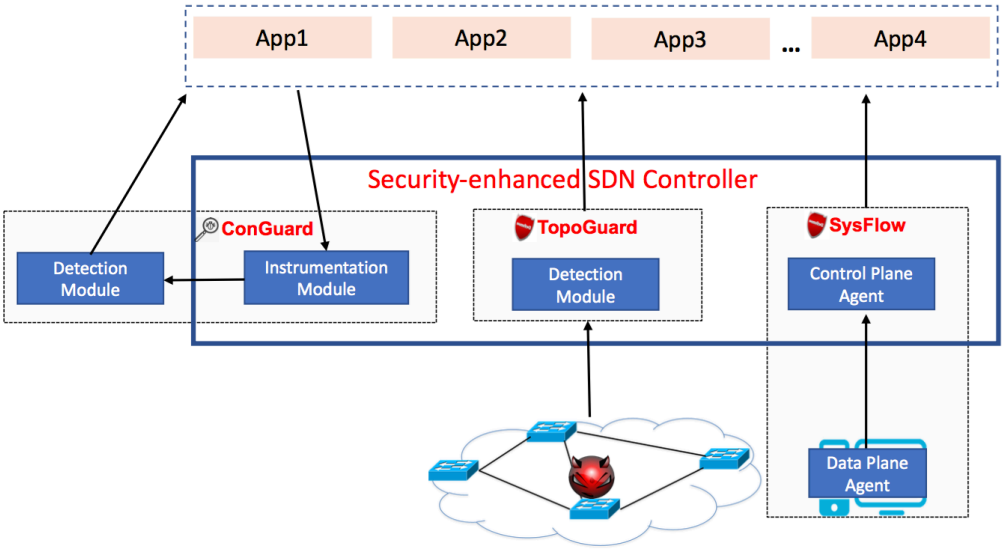


Figure 5.1: SDN security research framework.

First, the framework leverages the TOPOGUARD component to secure the network visibility in SDN. In particular, TOPOGUARD inspects control messages from data plane devices. Then, it adopts behavior analysis to detect and block those malicious messages related to topology poisoning attacks. In addition to mitigating topology poisoning attacks, this security component can be abstracted into a general threat detection engine to defend against more known and unknown network-side attacks against the SDN control plane. That is, we can feed SDN control messages to a detection engine, including known attack signature/pattern matching module or a machine learning based analytic module, to detect malicious/anomalous activities. Our successful experience

in the design and implementation of TOPOGUARD can help to develop such general network-side threat detection engine in the SDN control plane.

Moreover, the framework utilizes CONGUARD component to fortify network programmability by detecting and patching harmful race conditions. In particular, CONGUARD instruments the SDN controller and its applications to generate dynamic traces as an execution model. Based on the execution model, it further detects harmful race conditions in an offline manner. Then, the located vulnerabilities can guide developers or administrators to secure the SDN control plane by using online or offline patching. CONGUARD exhibits a general dynamic bug/vulnerability detection procedure in the SDN control plane. That is, we can reuse the instrumentation module to log critical operations into the execution trace and utilize detection module to automatically locate more different vulnerabilities, such as API misuses, data leakage, permission squatting, in either online or offline manner.

Besides, the SYSFLOW component can help the framework to protect modern infrastructures, e.g., cloud or enterprise, from emerging cyber attacks involving malicious system activities. In particular, SYSFLOW introduces a unified data plane agent in host systems and thus provide system security visibility and programmability to applications in SDN. Based on the insight of designing SYSFLOW, we can complement more security capabilities from other systems/components to the SDN control plane. For example, we can extend SDN with security functionality from virtual machines via introspection-based techniques [126, 127] to enable more fine-grained control on virtualized environments, e.g., cloud. Also, we can abstract the security capabilities from programmable host network stacks into the SDN control plane to provide unified packet processing logic for novel security applications.

6. CONCLUSION AND FUTURE WORK

Software-defined networking (SDN) technology has fundamentally change network infrastructures by providing two key innovations, i.e., holistic network visibility and flexible network programmability. Unfortunately, the security issues and limitations of those SDN-provided innovations is under-explored, which may impede the adoption of SDN to innovate programmable services in an infrastructure, e.g., enterprise or cloud. Motivated by the fact, we perform in-depth security analysis upon network visibility and programmability provided by SDN locate several security issues and limitations. First, we systematically study the security of SDN-provisioned network visibility stemmed from network topology management services/modules. From the study, we find several unknown security loopholes, which can incur severe topology poisoning attacks. Second, we conduct a systematic study on the network programmability provided by SDN. From the study, we notice that the concurrent programming model in SDN is vulnerable to harmful race conditions, which can further be exploited remotely to cause serious security and reliability issues in SDN. Third, the current SDN visibility and programmability only cover network-level information, which is far from enough to secure the entire infrastructure in today's enterprise/cloud systems. The reason lies in that existing SDN techniques fall short in detecting and preventing advanced cyber attacks involved in system-level malicious activities.

To mitigate those security issues, we present three solutions to enhance the security of the visibility and programmability provided by SDN. First, to mitigate topology poisoning attacks, we investigate possible defense strategies and present a new security extension on the SDN controller, called TOPOGUARD, that secures the topology management by providing light-weighted, automatic and real-time detection of topology poison attacks. Second, to prevent harmful race conditions in network programmability, we design and implement a dynamic framework, called CONGUARD, to effectively detect and validate those harmful race conditions in SDN controllers. By proactively locating and patching those concurrency vulnerabilities, this thesis can enhance the security of network programmability in SDN. Third, we introduce a novel flow-based model,

system flow, to abstract system activities and security capabilities. Upon the system flow model, we design and implement SYSFLOW, a framework to enable SDN controllers to effectively and efficiently enforce unified security intents with system security visibility and programmability.

In our future work, we plan to continue our research to make the SDN more secure and protect networked resources from emerging advanced attacks. We will conduct more security analyses on SDN architecture to detect and eliminate more vulnerabilities. For example, an SDN application may misuse service functions (which are also known as NBIs) to reference network state variables from other applications. However, verification and sanitization of more generalized uses of SDN control plane NBIs are still challenging issues. We plan to study these problems in future work. Moreover, we plan to leverage widely-used security analysis approaches, such as static/dynamic program analysis techniques, penetration testing techniques, and fuzzing testing techniques.

In the meantime, we plan to extend TOPOGUARD to support more SDN controllers. Also, we will complement CONGUARD with coverage-based techniques to increase its code coverage, such as symbolic execution [128, 129].

We will extend SYSFLOW in several aspects. First, we plan to extend SYSFLOW with a better programming abstraction, e.g., a domain-specific language (DSL), to support unified programmability between network flow model and system flow model and to hide tedious programming-language-related implementation details to a compiler and run-time system. Second, we plan to extend SYSFLOW to support more operating systems, e.g., Windows and Mac OS, and system events. Third, we plan to present more security designs to prevent attacks against SYSFLOW framework itself.

REFERENCES

[1] OpenFlow, "Innovate Your Network." http://www.openflow.org.

[2] Android, "Android Mobile Platform." https://www.android.com/.

[3] iOS, "iPhone OS." https://en.wikipedia.org/wiki/IOS.

[4] POX, "a python-based OpenFlow controller." https://noxrepo.github.io/pox-doc/html/.

[5] ryu, "ryu SDN framework." https://osrg.github.io/ryu/.

[6] Floodlight, "The Floodlight SDN Controller." http://www.projectfloodlight.org/floodlight/.

[7] OpenDaylight, "The OpenDaylight SDN Controller." https://www.opendaylight.org/.

[8] ONOS, "The ONOS SDN Controller." http://onosproject.org/.

[9] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch," in *SIGCOMM CCR'14*, 2014.

[10] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful Network-Wide Abstractions for Packet Processing," in *SIGCOMM'16*, 2016.

[11] 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 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[12] M. J. F. C. M. J. R. A. S. N. Foster, R. Harrison and D. Walker, "Frenetic: A Network Programming Language," in *In ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2011.

[13] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," in *USENIX NSDI*, 2013.

[14] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, *et al.*, "Hedera: dynamic flow scheduling for data center networks.," in *Nsdi*, vol. 10, 2010.

[15] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks.," in *Nsdi*, vol. 10, pp. 249–264, 2010.

[16] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *2013 Proceedings IEEE INFOCOM*, pp. 2211–2219, IEEE, 2013.

[17] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central control over distributed routing," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 43–56, ACM, 2015.

[18] S. Shin and G. Gu, "CloudWatcher: Network Security Monitoring Using OpenFlow in Dynamic Cloud Networks (or: How to Provide Security Monitoring as a Service in Clouds?)," in *Proceedings of the 7th Workshop on Secure Network Protocols (NPSec' 12), co-located with IEEE ICNP' 12*, October 2012.

[19] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 13)*, pp. 29–42, 2013.

[20] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha, "Software-defined latency monitoring in data center networks," in *International Conference on Passive and Active Network Measurement*, pp. 360–372, Springer, 2015.

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

[22] M. S. Kang, V. D. Gligor, and V. Sekar, "Spiffy: Inducing cost-detectability tradeoffs for persistent link-flooding attacks.," in *NDSS*, 2016.

[23] R. Braga, E. de Souza Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow.," in *LCN*, vol. 10, pp. 408–415, 2010.

[24] Y. Wang, Y. Zhang, V. K. Singh, C. Lumezanu, and G. Jiang, "Netfuse: Short-circuiting traffic surges in the cloud.," in *ICC*, pp. 3514–3518, Citeseer, 2013.

[25] "Floodlight Repo." https://github.com/floodlight/floodlight.

[26] "ONOS Repo." https://github.com/opennetworkinglab/onos.

[27] "OpenDaylight Repo." https://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/org/opendaylight/controller/distribution.opendaylight/.

[28] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *NDSS'15*, 2015.

[29] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the {SDN} control plane," in *26th USENIX Security Symposium (USENIX Security' 17)*, pp. 451–468, 2017.

[30] OpenFlow wireless, "n-casting using openflow." http://archive.openflow.org/wp/n-casting-mobility-using-openflow/.

[31] Mininet, "Rapid prototyping for software defined networks." http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/.

[32] Apache, "HTTP Server Project." https://httpd.apache.org/.

[33] Scapy, "Packet manipulation program." http://www.secdev.org/projects/scapy/.

[34] OpenFlow, "Openflow wireshark dissector." http://archive.openflow.org/wk/index.php/OpenFlow_Wireshark_Dissector.

[35] L. B. Kish, "Protection against the man-in-the-middle-attack for the kirchhoff-loop-johnson (-like)-noise cipher and expansion by voltage-based security," *Fluctuation and Noise Letters*, vol. 6, no. 01, pp. L57–L63, 2006.

[36] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *NSDI'10*, 2013.

[37] P. Kazemian, M. Chang, H. Zeng, S. Whyte, G. Varghese, and N. McKeown, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[38] P. P. D. K. M. Canini, D. Venzano and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[39] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A soft way for openflow switch interoperability testing," in *Proceedings of ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[40] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controller," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[41] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.

[42] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[43] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, August 2012.

[44] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: Transparent moving target defense using software defined networking," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*.

[45] S. Shin and G. Gu, "Attacking software-defined networks: A first feasibility study (short paper)," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013.

[46] 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 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[47] K. M. M. Dhawan, R. Poddar and V. Mann, "Cloudnaas: a cloud networking platform for enterprise applications," in *In proceedings of the 22th Annual Network & Distributed System Security Conference (NDSS'15)*, 2015.

[48] A. Ornaghi and M. Valleri, "Man In The Middle Attacks." http://www.blackhat.com/presentations/bh-europe-03/bh-europe-03-valleri.pdf.

[49] E. Jones and O. L. Moigne, "Ospf security vulnerabilities analysis," in *Internet-Draft draft-ietf-rpsec-ospf-vuln-02, IETF, June 2006*.

[50] D. G. G. Nakibly, A. Kirshon and D. Boneh, "Persistent ospf attacks," in *In proceedings of the 19th Annual Network & Distributed System Security Conference (NDSS'12)*, 2012.

[51] OLSR, "Optimized Link State Routing Protocol." http://www.ietf.org/rfc/rfc3626.txt.

[52] D. R. C. Adjih and P. MÃijhlethaler, "Attacks against olsr: Distributed key management for security," in *2005 OLSR Interop and Workshop*, July 2005.

[53] A. P. Y. Hu and D. B. Johnson, "Wormhole attacks in wireless networks," in *2005 OLSR Interop and Workshop*, July 2005.

[54] K. Benton, L. J. Camp, and C. Small, "Openflow vulnerability assessment," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*, August 2013.

[55] M. Casado, N. Foster, and A. Guha, "Abstractions for software-defined networks," *Commun. ACM*, vol. 57, pp. 86–95, Sept. 2014.

[56] A. EI-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev, "SDNRacer: Concurrency Analysis for Softeware-Defined Networks," in *PLDI'16*, 2016.

[57] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev, "SDNRacer: Detecting concurrency violations in software-defined networks," in *SOSR'15*, 2015.

[58] J. Huang, P. Meredith, and G. Rosu, "Maximal Sound Predictive Race Detection with Control Flow Abstract," in *PLDI'14*, 2014.

[59] C. Flanagan and S. Freund, "FastTrack: Efficient and Precise Dyanmic Race Detection," in *PLDI'09*, 2009.

[60] P. Maiya, A. Kanade, and R. Majumdar, "Race Detection for Android Applications," in *PLDI'14*, 2014.

[61] V. Raychev, M. Vechev, and M. Sridharan, "Effective Race Detection for Event-Driven Programs," in *OOPSLA'13*, 2013.

[62] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," in *Future Internet*, 2014.

[63] N. Weaver, R. Sommer, and V. Paxson, "Detecting Forged TCP Reset Packets," in *NDSS'09*, 2009.

[64] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency Attacks," in *USENIX Workshop on Hot Topics in Parallelism '12*, 2012.

[65] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System ," in *Communications of the ACM*, 1978.

[66] OpenFlow, "OpenFlow Specification 1.5." https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

[67] V. Kahlon and C. Wang, "Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs," in *CAV'10*, 2010.

[68] ASM, "Java Bytecode Analysis Framework." http://asm.ow2.org/.

[69] H. Database, "Java Graph Library." http://www.h2database.com/html/main.html.

[70] Cbench, "Scalable Benchmark for SDN Controllers." http://sourceforge.net/projects/cbench/.

[71] D. Tsafrir, T. Hertz, D. Wagner, and D. Silva, "Portably Solving File TOCTTOU Races with Hardness Amplification," in *FAST'08*, 2008.

[72] X. Cai, Y. Gui, and R. Johnson, "Exploiting Unix File-System Races via Algorithmic Complexity Attacks," in *S&P'09*, 2009.

[73] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, "Fixing Races for Fun and Profit: How to abuse atime," in *Usenix Security'05*, 2005.

[74] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," *TOCS'97*, 1997.

[75] C. Hsiao, J. Yu, S. Narayanasamy, and Z. Kong, "Race Detection for Event-Driven Mobile Applications," in *PLDI'14*, 2014.

[76] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race Detection for Web Applications," in *PLDI'12*, 2012.

[77] X. Sun, A. Agarwal, and T. S. E. Ng, "Attendre: Mitigating Ill Effects of Race Conditions in Openflow via Queueing Mechanism," in *ANCS '12*.

[78] K. Sen, "Race Directed Random Testing of Concurrent Programs," in *PLDI'08*, 2008.

[79] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *PLDI'09*, 2009.

[80] H. Mai, A. Khurishid, R. Agarwal, M. Caesar, P. Godfrey, and S. King, "Debugging the Data Plane with Anteater," in *SIGCOMM'11*, 2011.

[81] T. Ball, N. Bjorner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon: Towards Verifying Controller Programs in Software-defined Networks," in *PLDI'14*, 2014.

[82] C. Scott, A. Wundsam, B. Raghavan, A. Panda, J. L. A. Or, E. Huang, Z. Liu, A. EI-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *ATC'11*, 2011.

[83] A. Wu, S. S. D. Levin, and A. Feldmann, "Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences," in *SIGCOMM'14*, 2014.

[84] C. RÃűpke and T. Holz, "SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks," in *RAID'15*, 2015.

[85] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A Security Assessment Framework for Software-Defined Networks," in *NDSS'17*, 2017.

[86] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks," in *DSN'15*, 2015.

[87] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the Software-Defined Network Control Layer," in *NDSS'15*, 2015.

[88] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. Kang, "Rosemary: A Robust, Secure, and High-Performance Network Operating System," in *CCS'14*, 2014.

[89] C. RÂĺopke and T. Holz, "Retaining Control over SDN Network Services," in *NetSys'15*, 2015.

[90] T. Ball, "The Concept of Dynamic Analysis," in *FSE'99*, 1999.

[91] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *ISSTA'04*, 2004.

[92] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools," in *CAV'06*, 2006.

[93] Medjack, "Medjack to launch stepping-stone data ex-filtration." https://www.computerworld.com/article/2932371.

[94] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[95] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking," in *USENIX Conference on Security Symposium (SEC)*, 2018.

[96] H. Vijayakumar, J. Schiffman, and T. Jaeger, "Process Firewalls: Protecting Processes During Resource Access," in *ACM European Conference on Computer Systems (EuroSys)*, 2013.

[97] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "Jigsaw: Protecting Resource Access by Inferring Programmer Expectations," in *USENIX Conference on Security Symposium (SEC)*, 2014.

[98] A. BATES, D. J. TIAN, K. R. BUTLER, and MOYER, "Trustworthy whole-system provenance for the Linux kernel," in *USENIX Conference on Security Symposium (SEC)*, 2015.

[99] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integirty for Commodity OSes," in *ACM symposium on Operating systems principles (SOSP)*, 2007.

[100] N. L. P. Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor," in *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[101] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, " Ensuring Operating System Kernel Integrity with OSck," in *Architectural support for programming languages and operating systems (ASPLOS)*, 2011.

[102] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward Snoop-based Kernel integrity Monitor," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[103] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "KI-Mon: A Hardware-assisted Eventtriggered Monitoring Platform for Mutable Kernel Object," in *USENIX Conference on Security Symposium (SEC)*, 2013.

[104] V. Srinivasan, S. Suri, and G. Varghesea, "Packet Classification Using Tuple Space Search," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 1999.

[105] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, " Linux Security Module Framework ," in *Ottawa Linux Symposium (OLS)*, 2002.

[106] LMbench, "Tools for Performance Analysis." http://lmbench.sourceforge.net/.

[107] ab, "Apache HTTP server benchmarking tool." https://httpd.apache.org/docs/2.4/programs/ab.html.

[108] WGET Benchmark, "Benchmark for Linux wget tool." http://www.project-open.com/en/benchmark-wget.

[109] sysbench, "A modular, cross-platform and multi-threaded benchmark tool." http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html.

[110] X. J. et al., "Low-latency Networking: Where Latency Lurks and How to Tame It," in *arXiv:1808.02079*, 2018.

[111] "Estimating Log Generation for Security Information Event and Management." http://content.solarwinds.com/creative/pdf/Whitepapers/estimating_log_generation_white_paper.pdf.

[112] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and Elastic DDoS Defense," in *USENIX Security 15*, 2015.

[113] R. Braga and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow.,"

[114] A. Aydeger, N. Saputro, K. Akkaya, and M. Rahman, "Mitigating crossfire attacks using sdn-based moving target defense," in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pp. 627–630, IEEE, 2016.

[115] W. Han, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeymix: Toward sdn-based intelligent honeynet," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pp. 1–6, ACM, 2016.

[116] S. Kyung, W. Han, N. Tiwari, V. H. Dixit, L. Srinivas, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeyproxy: Design and implementation of next-generation honeynet via sdn," in *2017 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2017.

[117] X. Meng, Z. Zhao, R. Li, and H. Zhang, "An intelligent honeynet architecture based on software defined security," in *2017 9th International Conference on Wireless Communications and Signal Processing (WCSP)*, pp. 1–6, IEEE, 2017.

[118] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," July 2008.

[119] OpenDaylight, "Security Advisories of topology poisoning attacks in OpenDaylight." https://wiki.opendaylight.org/view/Security_Advisories.

[120] ONOS, "Secure Link Detection in ONOS."

[121] 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 *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 374–385, IEEE, 2018.

[122] T. Bui *et al.*, "Analysis of topology poisoning attacks in software-defined networking," 2015.

[123] S. Khan, A. Gani, A. W. A. 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, 2016.

[124] G. Lu, L. Xu, Y. Yang, and B. Xu, "Predictive analysis for race detection in software-defined networks," *Science China Information Sciences*, vol. 62, no. 6, p. 62101, 2019.

[125] 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 SIGSAC Conference on Computer and Communications Security*, pp. 3–16, ACM, 2018.

[126] T. Garfinkel, M. Rosenblum, *et al.*, "A virtual machine introspection based architecture for intrusion detection," Citeseer.

[127] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[128] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*, pp. 419–423, Springer, 2006.

[129] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with java pathfinder," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 97–107, ACM, 2004.