Attacking the Brain: Races in the SDN Control Plane

Lei Xu[1], Jeff Huang[1], Sungmin Hong[1], Jialong Zhang[1,2], and Guofei Gu[1]

[1]Texas A&M University, {xray2012,jeffhuang,ghitsh,guofei}@tamu.edu
[2]IBM Research, Jialong.Zhang@ibm.com

Abstract

Software-Defined Networking (SDN) has significantly enriched network functionalities by decoupling programmable network controllers from the network hardware. Because SDN controllers are serving as the brain of the entire network, their security and reliability are of extreme importance. For the first time in the literature, we introduce a novel attack against SDN networks that can cause serious security and reliability risks by exploiting harmful race conditions in the SDN controllers, similar in spirit to classic TOCTTOU (Time of Check to Time of Use) attacks against file systems. In this attack, even a weak adversary without controlling/compromising any SDN controller/switch/app/protocol but only having malware-infected regular hosts can generate external network events to crash the SDN controllers, disrupt core services, or steal privacy information. We develop a novel dynamic framework, ConGuard, that can effectively detect and exploit harmful race conditions. We have evaluated ConGuard on three mainstream SDN controllers (Floodlight, ONOS, and OpenDaylight) with 34 applications. ConGuard detected totally 15 previously unknown vulnerabilities, all of which have been confirmed by developers and 12 of them are patched with our assistance.

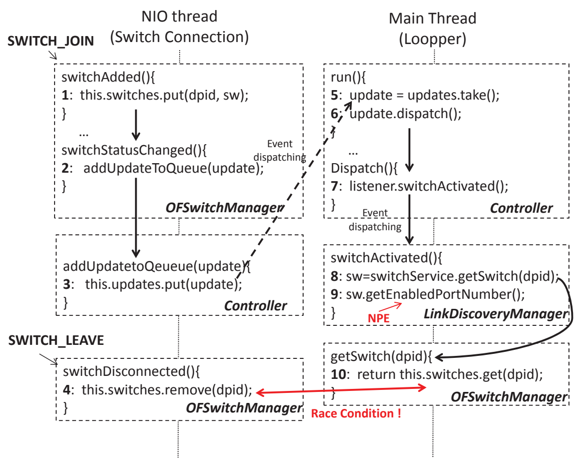
1 Introduction

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 [16].

Despite the popularity, unfortunately, SDN has also changed the attack surface of traditional networks. An SDN controller and its applications maintain 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 paper, we present a new attack, namely *state manipulation attack*, in the SDN control plane that is rooted in the asynchronism of SDN. The asynchronism leads to many harmful race conditions on the shared network states, which can be exploited by the attackers to cause denial of services (e.g., controller crash, core service disruption) and privacy leakage, etc. On the surface, this is similar to the well-known TOCTTOU (Time of Check to Time of Use) attacks [46, 14, 12] against file systems. However, this attack is closely tied to the unique SDN semantics, which makes all popular SDN controllers (e.g., Floodlight [1], ONOS [3], and OpenDaylight [4]) vulnerable. Consider a real example we discovered in the Floodlight controller in Figure 1. 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 LinkDiscoveryManager 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 condi-



```
switchAdded(){
1:  this.switches.put(dpid, sw);
}
  ...
switchStatusChanged(){
2:   addUpdateToQueue(update);
}

addUpdatetoQeueue(update){
3:   this.updates.put(update);
}

switchDisconnected(){
4:  this.switches.remove(dpid);
}

run(){
5:  update = updates.take();
6:  update.dispatch();
}
  ...
Dispatch(){
7:  listener.switchActivated();
}

switchActivated(){
8: sw=switchService.getSwitch(dpid);
9: sw.getEnabledPortNumber();
}

getSwitch(dpid){
10:  return this.switches.get(dpid);
}
```

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

tions 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 1.

The key insight of State Manipulation Attack is that we can leverage the existence of such harmful race conditions in SDN controllers to trigger inconsistent network states. Nevertheless, a successful attack requires tackling two challenging problems:

- First, how to locate such harmful race conditions in the SDN controller source code?

- Second, how to trigger the harmful race conditions by an external attacker who has no control of the controller schedule?

For the first problem, the key challenges are that it is generally unknown if a race condition is harmful or not, and that detecting race conditions in a program is generally undecidable. Although many data race detectors have been developed for different domains [18, 32, 22, 19, 31, 36], 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. For example, a data race detector cannot find the race condition in Figure 1 because the accesses to the *switches* variable are all protected by synchronization. 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 this 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.

For the second problem, the key challenge is that a harmful race condition occurs very rarely in normal operations, but relies on a combination of a certain input and an unexpected thread schedule to manifest. As the adversary typically has no control of the machine or operating system running the SDN controllers, even if a harmful race condition is known, it is difficult for an adversary to create the input and schedule combination to trigger the harmful race condition.

Nevertheless, we show that an adversary can remotely exploit many harmful race conditions with a high success ratio by injecting the "right" external events into the SDN network. Because SDN controllers define an event handler to process each network event, a correlation between external network events and their corresponding event handlers can be established by analyzing the controller source code. By further mapping the event handlers to their operations, we can correlate the conflicting operations in a harmful race condition to their corresponding network events. An adversary can then generate many sequences of these network events repeatedly to increase the chance of hitting a right schedule to trigger the harmful race condition.

We have designed and implemented a framework called ConGuard for exploiting concurrency vulnerabilities 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[1] 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 be triggered by asynchronous network events in unexpected schedules.
- We design CONGUARD, a novel framework to pinpoint and exploit 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. By the time of writing, we have already assisted the developers to patch 12 of them.

The rest of the paper is organized as follows: Section 2 introduces background. Section 3 discusses the state manipulation attack. Section 4 and Section 5 describe the design and implementation of our CONGUARD framework. Section 6 evaluates CONGUARD. Section 7 discusses defense mechanisms to mitigate this kind of attacks. Section 8 discusses limitations of our approach and future work. Section 9 reviews related work and Section 10 concludes this paper.

2 Background

In this section, we introduce the necessary background of SDN in order to understand the harmful race conditions in this domain.

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, and access control. OpenFlow [6] is the dominant communication protocol between the SDN control plane and the data plane. In this paper, we may use SDN and OpenFlow interchangeably.

[1]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 [6, 13].

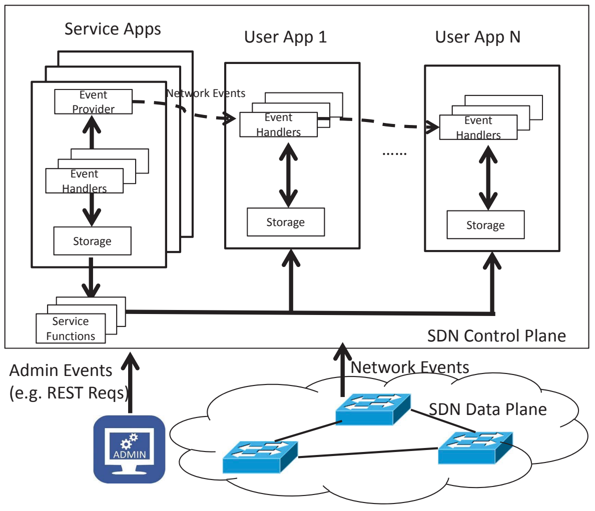


Figure 2: The abstraction model of the SDN control plane .

The SDN control plane embraces a concurrent modular model. As shown in Figure 2, 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 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[2] 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 paper.

Table 1 shows several network-related events and administrative events in the SDN control plane. In this paper, we focus on these network events because they are

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

Table 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

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 4.1.1.

3 State Manipulation Attacks

In this section, we present state manipulation attacks in SDN networks by exploiting harmful race conditions. We first present the threat model and explain how an external adversary can generate various network events in an SDN network. We then discuss two vulnerabilities related to harmful race conditions that we discovered in existing SDN controllers, and we show how an attacker can exploit them to steal privacy information and disrupt important services of SDN networks. We will discuss more vulnerabilities found in our experiments in Section 6.

3.1 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 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. As we discuss in Section 6.5, 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.

3.2 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[3], which can trigger `OFP_PACKET_IN` events. Switch port events (i.e., `PORT_UP` and `PORT_DOWN`) can also be indirectly generated by network 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 [49] or drop control messages to disrupt the connection to generate `SWITCH_LEAVE`, thereby incurring `SWITCH_JOIN` subsequently. For example, as shown in Figure 3, we can use TCP reset to generate a `SWITCH_LEAVE` event in the Floodlight controller.

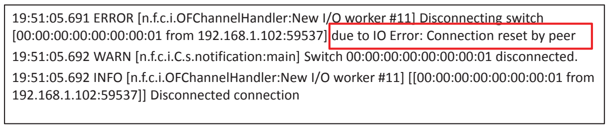

```
19:51:05.691 ERROR [n.f.c.i.OFChannelHandler:New I/O worker #11] Disconnecting switch
[00:00:00:00:00:00:00:01 from 192.168.1.102:59537] due to IO Error: Connection reset by peer
19:51:05.692 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 disconnected.
19:51:05.692 INFO [n.f.c.i.OFChannelHandler:New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

Figure 3: `SWITCH_LEAVE` event generated by TCP Resets.

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

3.3 Attack Cases

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.

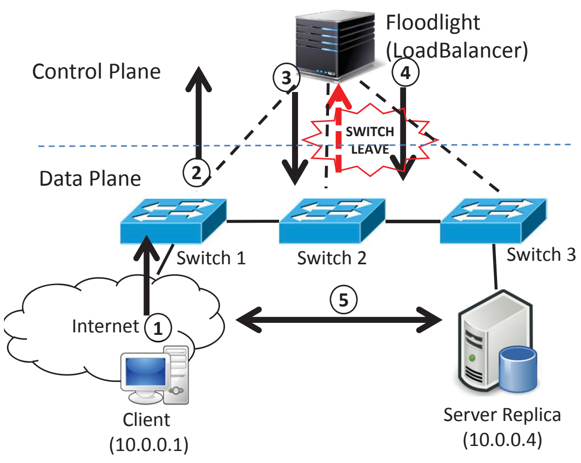


Figure 4: Attacking the Floodlight LoadBalancer.

3.3.1 Stealing Privacy Information

Figure 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. In addition, 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. We detail more about the exploitation of such vulnerability in Section 6.6.

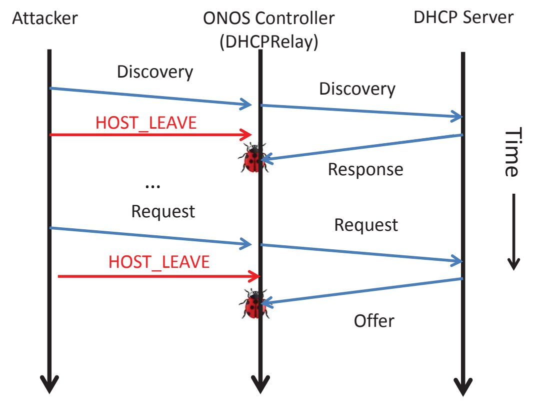


Figure 5: Attacking the ONOS DHCPRelay application.

3.3.2 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 `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 5. 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 detail more about such attack in Section 6.6.

4 ConGuard Overview

In this section, we present our framework, ConGuard, for detecting and exploiting the race condition vulnerabilities in SDN controllers. ConGuard contains two main phases: (i) locating harmful race conditions in the controller source code by utilizing dynamic analysis and adversarial state racing, (ii) triggering harmful race conditions in the running SDN controller by remotely injecting right external network events with the proper timing.

4.1 Pinpointing Harmful Race Conditions

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 1, 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 1 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.

4.1.1 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 paper, we utilize happens-before relations [28] 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 paper. 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 6, we list happens-before relations we derive in the SDN context by studying implementations of SDN controllers and OpenFlow switch specification [5]. 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 6, 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 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

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

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

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

Singleton Task

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

Figure 6: Happens-before rules in the SDN control plane.

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.

4.1.2 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 4.1.1. 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 [19] 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 [24, 31] 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 facilitate 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.

4.1.3 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.

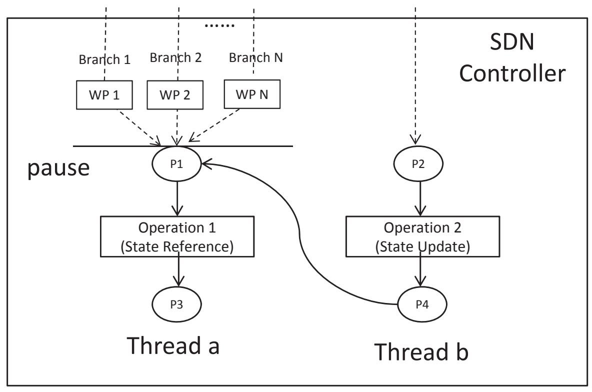


Figure 7: *Active Scheduling* to force a state update to execute before a state reference (WP denotes waypoint).

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 1) to follow a specific erroneous order, as shown in Figure 7. 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 *switches.getSwitch(dpid)* in Figure 1). 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.

4.2 Remotely Triggering 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 1, 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.

4.2.1 Trigger Correlation

Since SDN controllers define different handler functions to process various 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 1, the dpid of the *update event* `SWITCH_LEAVE` should be consistent with that of the *trigger event* `SWITCH_JOIN`.

4.2.2 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 1, 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. We present more examples later in Table 5.

Moreover, an attacker can tactically increase the probability of success by selecting a larger vulnerable window [51] for a specific exploitation. The vulnerable window is the timing window that a concurrency vulnerability may occur. For some vulnerabilities, we found that their vulnerable 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 5, the attacker can launch the attack when the network delay is high. In such a case, an attacker can first utilize a probe testing to pick up an advantageous condition to launch the attack.

5 Implementation

We have implemented CONGUARD and tested it on three mainstream SDN controllers, including Floodlight [1], ONOS [3] and OpenDaylight [4].

Input Generation: To inject network events, we introduce an SDN control plane specific input generator in our framework. We utilize Mininet 2.2 [7], an SDN network simulator, to mock an SDN testbed. Mininet can generate all the network events as shown in Table 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 [9] 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 [2]). 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 2) with the identifier of the name of the class.

Table 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.

6 Evaluation

In this section, we present our evaluation results of CONGUARD on the three mainstream open-source SDN controllers with 34 applications as listed in Table 7 in Appendix A. 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.

Table 3: Overall race detection results. (#RT: the size of raw traces before preprocessing; #OT: the size of optimized traces; RE: reduction ratio by preprocessing; OTATime: the total time for offline trace analysis; #Races: the number of detected race conditions; #RSVs: the number of Race State Variables)

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

6.1 Detection Results

Table 3 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 CONGUARD with Floodlight version 1.1: (1) enforces only thread-based happens-before rules; (2) enforces both thread-based and SDN-specific 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 conducting more precise concurrency defect detection in SDN controllers.

6.2 Comparing With Existing Techniques

To evaluate the effectiveness of our approach for identifying harmful race conditions, we also compared CONGUARD with an SDN-specific race detector, SDNRacer [18], and a state-of-the-art general dynamic race detector, RV-Predict (version 1.7) [22].

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.

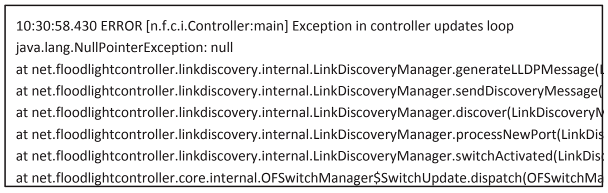
Comparing with RV-Predict. RV-Predict is the state-of-the-art general-purpose data race detector that achieves maximal detection capability based on a program trace but does not consider harmful race conditions, and does not have SDN-specific causality rules. We evaluated RV-Predict as a Java agent for 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[4]. The reason is that all those harmful race conditions are caused by well-synchronized operations in Java concurrent libraries, which are not data races.

6.3 CONGUARD Runtime Performance

We evaluated the runtime performance of CONGUARD for trace collection using Cbench [8], 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 CONGUARD 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

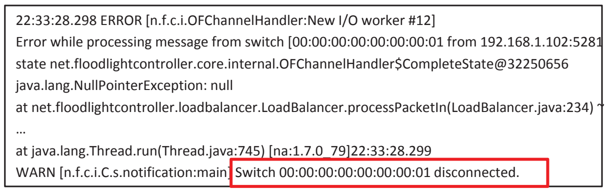
[4] We manually backtracked the call graph information for every data race reported by RV-Predict and checked if it could lead to harmful race conditions.

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.



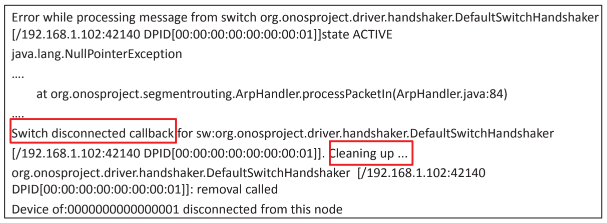
```
10:30:58.430 ERROR [n.f.c.i.Controller:main] Exception in controller updates loop
java.lang.NullPointerException: null
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.generateLLDPMessage(L
at net.floodlightcontroller.linkdiscovery.internal.LinkDiscoveryManager.sendDiscoveryMessage(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

Figure 8: A harmful race condition causes the Floodlight controller out of service.



```
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 9: A harmful race condition in Floodlight causes disconnection of a switch.



```
Error while processing message from switch org.onosproject.driver.handshaker.DefaultSwitchHandshaker
[/192.168.1.102:42140 DPID[00:00:00:00:00:00:00:01]]state ACTIVE
java.lang.NullPointerException
....
      at org.onosproject.segmentrouting.ArpHandler.processPacketIn(ArpHandler.java:84)
....
Switch disconnected callback for sw:org.onosproject.driver.handshaker.DefaultSwitchHandshaker
[/192.168.1.102:42140 DPID[00:00:00:00:00:00:00:01]]. Cleaning up ...
org.onosproject.driver.handshaker.DefaultSwitchHandshaker  [/192.168.1.102:42140
DPID[00:00:00:00:00:00:00:01]]: removal called
Device of:0000000000000001 disconnected from this node
```

Figure 10: A harmful race condition in ONOS causes disconnection of a switch.

6.4 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 4. We 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 scheduling (as shown in Figure 8) 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 9 and Figure 10 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[5]. 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** 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

[5]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.

Table 4: Summary of harmful race conditions uncovered by CONGUARD. Impact #1: System Crash; Impact #2: Connection Disruption; Impact #3: Service Disruption; Impact #4: Service Chain Interference.

Controller	Application	Bug#	Correlated Attack Event Pairs <trigger event, update event>	Impact Vector			
				#1	#2	#3	#4
Flood-light	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>		●	●	●
		$7^\dagger$	<OFP_PACKET_IN, REST_REQUEST>		●	●	●
		$8^\dagger$	<OFP_PACKET_IN, REST_REQUEST>		●	●	●
		$9^\dagger$	<OFP_PACKET_IN, REST_REQUEST>		●	●	
	Statistics	$10^\dagger$	<REST_REQUEST, SWITCH_LEAVE>			●	
ONOS	SegmentRouting	11	<OFP_PACKET_IN, HOST_LEAVE>		●	●	●
	DHCPRelay	12	<OFP_PACKET_IN, HOST_LEAVE>		●	●	●
OpenDay-light	Host Tracker	$13^\dagger$	<REST_REQUEST, HOST_LEAVE>			●	
		14	<HOST_JOIN, HOST_LEAVE>				●
	Web UI	$15^{\dagger *}$	<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

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.

6.5 Remote Exploitation Analysis

We consider all of the detected harmful race conditions can be triggered non-deterministically in normal operations of an SDN/OpenFlow network. In addition, we study the adversarial exploitations of those harmful race conditions by a remote attacker as discussed in Section 3.1. We first investigate their external triggers, i.e., the *trigger event* and *update event* pair, as shown in Table 4. For 15 harmful race conditions we detected, we found 9 of them can be exploited by external network events. An attacker with the control of compromised hosts/virtual machines in SDN networks can easily trigger three harmful race conditions (i.e., **Bug-11**, **Bug-12** and **Bug-14**) by generating `OFP_PACKET_IN`, `HOST_JOIN`, `HOST_LEAVE`, `PORT_UP`, and `PORT_DOWN`. Moreover, the attacker can remotely exploit 6 more harmful race conditions (i.e., **Bug-1**, **Bug-2**, **Bug-3**, **Bug-4**, **Bug-5** and **Bug-6**) by utilizing `SWITCH_JOIN` and `SWITCH_LEAVE` events when the SDN network utilizes in-band control messages. For the rest 6 harmful race conditions (i.e., **Bug-7**, **Bug-8**, **Bug-9**, **Bug-10**, **Bug-13**, and **Bug-15**), we found that they correlate with REST API requests which are administrative events and might be protected by TLS/SSL. We consider the exploitation of those 6 harmful race conditions is out of scope of the paper since we do not assume an attacker can generate authenticated administrative events in the paper. Also, we found that there might have multiple triggers for a specific harmful race condition since SDN applications may reference the same network state variable in order to react upon various network events.

Moreover, based on results from Table 4, we evaluate the feasibility of an external attacker to exploit harmful race conditions. In particular, we utilize Mininet to inject ordered attack event sequences with a proper timing and test how many trials an external attacker needs to trigger a harmful race condition. Table 6 shows the average number of injected event sequences from 5 successful exploitations for an attacker to exploit a harmful race condition in an SDN controller[6]. Consequently, we found an attacker can exploit 7 out of 9 harmful race conditions within only hundreds of attempts.

Furthermore, Table 5 lists some feedback information that an attacker can use to infer the result of exploitations. For **Bug-1**, **Bug-2**, **Bug-3**, and **Bug-4**, the attacker can infer the failure of exploitation by monitoring LLDP packets from the SDN controller to the active ports of the activated switch. For **Bug-5** and **Bug-6**, the attacker can notice the unsuccessful exploitations by receiving re-

[6]Note that since some attack event sequence may trigger multiple harmful race condition (e.g., <`SWITCH_LEAVE`, `SWITCH_JOIN`> can trigger **Bug-1**, **Bug-2**, **Bug-3**, and **Bug-4**), we only record the first bug exploitation because an exploitation of harmful race condition may disrupt the operation of the SDN controller.

sponses from the virtual IP address of the public service. For **Bug-12**, as long as the attacker receives a DHCP response/offer message, he/she can infer that the exploitation fails. More importantly, the indicative information is useful for the attacker to tune their exploitations such as to minimize the timing interval between *trigger event* and *update event*.

In addition to injecting ordered attack events and tuning the timing between attack events, we also found that, the vulnerable windows of 7 harmful race conditions (i.e., **Bug-1**, **Bug-2**, **Bug-3**, **Bug-4**, **Bug-5**, **Bug-6**, and **Bug-12**) can be enlarged in some conditions. In particular, the vulnerable windows of **Bug-1** and **Bug-4** include the dispatch of all previous updates of Floodlight controller as shown in Figure 1, where the more unprocessed network events (e.g., `SWITCH_JOIN`, `PORT_UP`, and `PORT_DOWN`) and the more event handler functions of SDN applications can enlarge the window. The vulnerable windows of **Bug-2** and **Bug-3** are linearly correlated with the numbers of active ports of the switch. The vulnerable windows of **Bug-5** and **Bug-6** are relevant to the number of switches in the route between the compromised host and the target server in Figure 4. Lastly, as discussed in Section 3.3.2, the vulnerable window of **Bug-12** is subject to round-trip delay between ONOS controller and the DHCP server. An attacker could utilize them to increase the success rate of exploitation.

Table 5: Feedback information for the exploitations of harmful race conditions.

Bug #	Indications of Failed Exploitation
1,2,3,4	receipt of LLDP packets
5,6	receipt of responses from the service IP address
12	receipt of DHCP response/offer messages

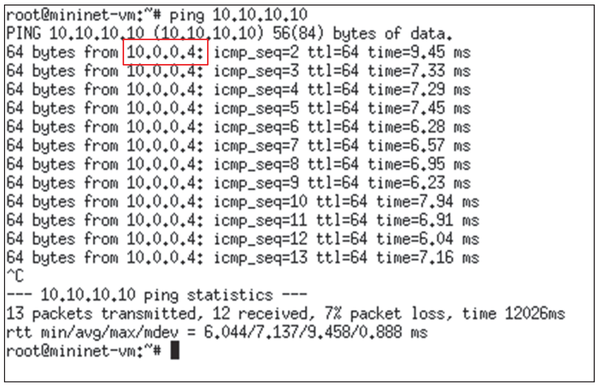
Table 6: Remote exploitation result.

Bug #	Attack Case	Trials (average)
1	(`SWITCH_JOIN`,`SWITCH_LEAVE`)	10.6
2	(`SWITCH_JOIN`,`SWITCH_LEAVE`)	78.4
3	(`SWITCH_JOIN`,`SWITCH_LEAVE`)	120
4	(`SWITCH_JOIN`,`SWITCH_LEAVE`)	10
5	(`OFP_PACKET_IN`,`SWITCH_LEAVE`)	67.6
6	(`OFP_PACKET_IN`,`SWITCH_LEAVE`)	106.8
11	(`OFP_PACKET_IN`,`HOST_LEAVE`)	-
12	(`OPP_PACKET_IN`,`HOST_LEAVE`)	1
14	(`HOST_LEAVE`,`HOST_JOIN`)	-

6.6 Case Studies

Here we detail two state manipulation attack examples as briefly introduced in Section 3.3.

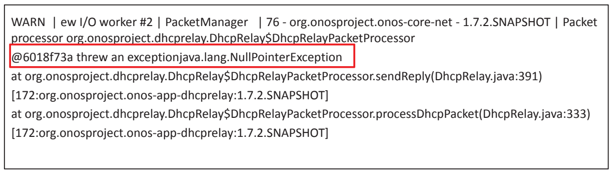
Sniffing Physical IP Address of Service Replica. In order to exploit the harmful race condition remotely, we set up an experiment as shown in Figure 4 in Mininet [7]. To launch the attack, we periodically injected `OFP_PACKET_IN` and `SWITCH_LEAVE` events. In particular, we updated the source IP address 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 11 below.



```
root@mininet-vm:~# ping 10.10.10.10
PING 10.10.10.10 (10.10.10.10) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=9.45 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=7.33 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=7.29 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=64 time=7.45 ms
64 bytes from 10.0.0.4: icmp_seq=6 ttl=64 time=6.28 ms
64 bytes from 10.0.0.4: icmp_seq=7 ttl=64 time=6.57 ms
64 bytes from 10.0.0.4: icmp_seq=8 ttl=64 time=6.95 ms
64 bytes from 10.0.0.4: icmp_seq=9 ttl=64 time=6.23 ms
64 bytes from 10.0.0.4: icmp_seq=10 ttl=64 time=7.94 ms
64 bytes from 10.0.0.4: icmp_seq=11 ttl=64 time=6.91 ms
64 bytes from 10.0.0.4: icmp_seq=12 ttl=64 time=6.04 ms
64 bytes from 10.0.0.4: icmp_seq=13 ttl=64 time=7.16 ms
^C
--- 10.10.10.10 ping statistics ---
13 packets transmitted, 12 received, 7% packet loss, time 12026ms
rtt min/avg/max/mdev = 6.044/7.137/9.458/0.888 ms
root@mininet-vm:~# █
```

Figure 11: Privacy leakage in Floodlight LoadBalancer.

Disrupting Packet Processing Service. 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 12) 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 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 12: Service disruption in ONOS DHCPRelay.

7 Defense Schemes

In this section, we discuss some possible defense techniques that developers or network administrators can use to mitigate this type of attacks.

Safety Check. To defend against the attack, one way is to remove those harmful race conditions once detected. The root cause of harmful race conditions is the concurrency violations inside the SDN controller/applications that may render inconsistency during state transition. For example, a concurrent `SWITCH_LEAVE` event modifying the state of a switch may incur some logic flaw in the handler of `SWITCH_JOIN` event for the switch. In this paper, we mitigate the exploitation of harmful race conditions by adding extra state checks in the SDN controller/applications to ensure the state is unchanged at the referenced location. By adding such safety checks, we have assisted the developers of SDN controllers to patch 12 harmful race conditions. Our future work will investigate how to automate this procedure.

Deterministic Execution Runtime. Another defense solution is to guarantee the deterministic execution of state operations in the SDN control plane at runtime. However, such a solution is difficult to correctly implement due to the undecidable order of two race operations. Even though we successfully resolve the orders between race operations, it inevitably undermines the parallelism of event processing, which further affects the overall performance of SDN controllers for a large-scale network environment. Designing a deterministic execution runtime system to mitigate concurrency errors in the SDN control plane with minor performance overhead is a meaningful future research direction.

Sanitizing External Events. One important factor of successful exploitation of harmful race conditions lies in that an attacker can intentionally inject various control plane messages (e.g., `HOST_LEAVE`, `SWITCH_LEAVE`) to modify the internal state inside the SDN control plane. In this sense, adopting an anomaly detection system to sanitize suspicious state update events could impede the exploitation of harmful race conditions. For example, an anomaly detection system may block some host to join SDN networks if its connection status is flipping frequently in a short time. Designing such anomaly detection with low false positives/negatives is worth future investigation.

8 Limitations and Discussion

Testing Coverage. As a common drawback of dynamic analysis techniques [10], the race detection part of CONGUARD cannot cover all execution paths. Thus, CONGUARD 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 CONGUARD with other coverage-based techniques such as symbolic execution [47, 42].

Supporting More Controllers and Other Event-driven Systems. The current implementations of CONGUARD 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 CONGUARD is based on the abstracted semantics of the SDN control plane. In that sense, we can easily port CONGUARD 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 CONGUARD with precise domain-specific models (like happens-before rules discussed in Section 4.1.1) 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.

9 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 [46, 14, 12]. In this paper, 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 paper 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 [39, 19, 22] and domain-specific programs in web and Android [21, 31, 36, 33]. However, these existing detectors do not work well for harmful race conditions discussed in this paper 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 [32, 18] 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 [45] 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 [41, 23], 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 [30] presents a static analysis approach to debug SDN data plane by translating network invariant verification to the boolean satisfiability problem. NICE [15] complements model checking with symbolic execution to locate operation bugs inside SDN controller applications. Vericon [11] develops a system to verify if an SDN program is correct to user-specified admissible network topologies and desired network-wide invariants. OFRewind [40] proposes to reproduce SDN operation errors by utilizing record-and-replay technique. SOFT [27] complements symbolic execution with cross checking to test interoperability of SDN switches. STS [50] leverages delta debugging algorithm to derive minimal causal sequence for SDN controller operation bugs, which can facilitate network troubleshooting and root-cause analysis. Veriflow [26] proposes a shim layer between the SDN controller and switches to check network invariants. NetPlumber [25] 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 [38]. DELTA [29] presents a fuzzing-based penetration testing framework to find unknown attacks in SDN controllers. TopoGuard [20] 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 paper we study a new threat to the SDN, i.e., harmful race conditions in the SDN control plane.

To fortify SDN networks, AvantGuard [44] and FloodGuard [48] propose schemes to defend against unique Denial-of-Service attacks inside SDN networks. FortNOX [35] and SE-FloodLight [34] propose several security extensions to prevent malicious applications from violating security policies enforced in the data plane. SPHINX [17] presents a novel model representation, called flow-graph, to detect several network attacks against SDN networks. Rosemary [43] and [37] 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 paper.

10 Conclusion

In this work, 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 exploiting harmful race conditions. Our tool ConGuard has found 15 previously unknown vulnerabilities in three mainstream SDN controllers. We hope this work will pave a foundation for detecting concurrency vulnerabilities in the SDN control plane, and in general will stimulate more future research to improve SDN security.

Acknowledgements

We want to thank our shepherd William Enck and the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the the National Science Foundation (NSF) under Grant no. 1617985, 1642129, and 1700544, and a Google Faculty Research award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and Google.

References

[1] Floodlight Repo. `https://github.com/floodlight/floodlight`.

[2] Java Graph Library. `http://www.h2database.com/html/main.html`.

[3] ONOS Repo. `https://github.com/opennetworkinglab/onos`.

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

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

[6] OpenFlow Specification v1.4.0. `http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf`.

[7] Rapid prototyping for software defined networks. `http://mininet.org/`.

[8] Scalable Benchmark for SDN Controllers. `http://sourceforge.net/projects/cbench/`.

[9] ASM. Java Bytecode Analysis Framework. `http://asm.ow2.org/`.

[10] Ball, T. The Concept of Dynamic Analysis. In *FSE'99* (1999).

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

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

[13] Braun, W., and Menth, M. Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices. In *Future Internet* (2014).

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

[15] Canini, M., Venzano, D., Peresini, P., Kostic, D., and Rexford, J. A NICE Way to Test OpenFlow Applications. In *NSDI'12* (2012).

[16] Casado, M., Foster, N., and Guha, A. Abstractions for software-defined networks. *Commun. ACM 57*, 10 (Sept. 2014), 86–95.

[17] Dhawan, M., Poddar, R., Mahajan, K., and Mann, V. SPHINX: Detecting security attacks in software-defined networks. In *NDSS'15* (2015).

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

[19] Flanagan, C., and Freund, S. FastTrack: Efficient and Precise Dyanmic Race Detection. In *PLDI'09* (2009).

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

[21] Hsiao, C., Yu, J., Narayanasamy, S., and Kong, Z. Race Detection for Event-Driven Mobile Applications. In *PLDI'14* (2014).

[22] Huang, J., Meredith, P., and Rosu, G. Maximal Sound Predictive Race Detection with Control Flow Abstract. In *PLDI'14* (2014).

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

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

[25] Kazemian, P., Chang, M., Zeng, H., Whyte, S., Varghese, G., and McKeown, N. Real Time Network Policy Checking using Header Space Analysis. In *NSDI'13* (2013).

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

[27] Kuzniar, M., Peresini, P., Canini, M., Venzano, D., and Kostic, D. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT'12* (2012).

[28] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System . In *Communications of the ACM* (1978).

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

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

[31] Maiya, P., Kanade, A., and Majumdar, R. Race Detection for Android Applications. In *PLDI'14* (2014).

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

[33] Petrov, B., Vechev, M., Sridharan, M., and Dolby, J. Race Detection for Web Applications. In *PLDI'12* (2012).

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

[35] Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., and Gu, G. A Security Enforcement Kernel for OpenFlow Networks. In *HotSDN'12* (August 2012).

[36] Raychev, V., Vechev, M., and Sridharan, M. Effective Race Detection for Event-Driven Programs. In *OOPSLA'13* (2013).

[37] ROPKE, C., AND HOLZ, T. Retaining Control Over SDN Network Services. In *NetSys'15* (2015).

[38] RPKE, C., AND HOLZ, T. SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In *RAID'15* (2015).

[39] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS* (1997).

[40] SCOTT, C., WUNDSAM, A., RAGHAVAN, B., PANDA, A., A. OR, J. L., HUANG, E., LIU, Z., EI-HASSANY, A., WHITLOCK, S., ACHARYA, H., ZARIFIS, K., AND SHENKER, S. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *ATC'11* (2011).

[41] SEN, K. Race directed random testing of concurrent programs. In *PLDI'08* (2008).

[42] SEN, K., AND AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV'06* (2006).

[43] SHIN, S., SONG, Y., LEE, T., LEE, S., CHUNG, J., PORRAS, P., YEGNESWARAN, V., NOH, J., AND KANG, B. Rosemary: A Robust, Secure, and High-Performance Network Operating System. In *CCS'14* (2014).

[44] SHIN, S., YEGNESWARAN, V., PORRAS, P., AND GU, G. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *CCS'13* (2013).

[45] SUN, X., AGARWAL, A., AND NG, T. S. E. Attendre: Mitigating Ill Effects of Race Conditions in Openflow via Queueing Mechanism. In *ANCS '12*.

[46] TSAFRIR, D., HERTZ, T., WAGNER, D., AND SILVA, D. Portably Solving File TOCTTOU Races with Hardness Amplification. In *FAST'08* (2008).

[47] VISSER, W., PĂSĂREANU, C. S., AND KHURSHID, S. Test input generation with java pathfinder. In *ISSTA'04* (2004).

[48] WANG, H., XU, L., AND GU, G. FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks. In *DSN'15* (2015).

[49] WEAVER, N., SOMMER, R., AND PAXSON, V. Detecting Forged TCP Reset Packets. In *NDSS'09* (2009).

[50] WU, A., D. LEVIN, S. S., AND FELDMANN, A. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *SIGCOMM'14* (2014).

[51] YANG, J., CUI, A., STOLFO, S., AND SETHUMADHAVAN, S. Concurrency Attacks. In *n USENIX Workshop on Hot Topics in Parallelism '12* (2012).

A Tested SDN Applications

Table 7: 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