SDNRacer: Concurrency Analysis for Software-Defined Networks

Ahmed El-Hassany[‡] Jeremie Miserez[♯] Pavol Bielik[♯]
Laurent Vanbever[‡] Martin Vechev[♯]

Dept. of Information Technology and Electrical Engineering[‡] Dept. of Computer Science[♯]
ETH Zürich, Switzerland

{eahmed,lvanbever@ethz.ch} miserezj@student.ethz.ch {pavol.bielik, martin.vechev@inf.ethz.ch}

http://sdnracer.ethz.ch

Abstract

Concurrency violations are an important source of bugs in Software-Defined Networks (SDN), often leading to policy or invariant violations. Unfortunately, concurrency violations are also notoriously difficult to avoid, detect and debug.

This paper presents the design and the implementation of a sound and complete dynamic analyzer, SDNRacer, which can ensure a network is free of harmful errors such as data races or per-packet incoherences. SDNRacer is based on two key ingredients: *(i)* a precise happens-before model for SDNs that captures when events can happen concurrently, and; *(ii)* a set of sound, domain-specific filters that reduce the reported violations by orders of magnitude.

We evaluated SDNRacer on several real-world SDN controllers, running both reactive and proactive applications in large networks. We show that SDNRacer is practically effective: it quickly (within 30 seconds in 90% of the cases) pinpoints harmful concurrency violations (including unknown bugs) without overwhelming the user with false positives.

Categories and Subject Descriptors C.2.3 [*Computer-Communication Networks*]: Network Operations; D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Software Defined Networking, OpenFlow, Commutativity Specification, Happens-before, Nondeterminism

1. Introduction

In the last few years, Software-Defined Networking (SDN) managed to establish itself as a promising approach for designing and operating computer networks. At its core, SDN is predicated around two key principles. First, SDN argues for a physical separation between the *control-plane*, which decides how to forward data packets, and the *data-plane*, which forwards packets according to control-plane decisions. Second, SDN argues for a (logical) centralization of the control logic which relies on standardized APIs, such as OpenFlow [1], to program forwarding state in each network device (SDN switch).

While the basic premises of SDN are simple, realizing this vision in practice requires developers to build highly sophisticated and reliable control software operating on top of a network—a highly asynchronous and distributed environment. Building such highly asynchronous programs is known to be a very difficult problem due to inadvertently introducing harmful concurrency errors.

In the context of SDN, there are two places where concurrent interference can occur: *(i)* within the SDN control software itself (*e.g.*, if it is multi-threaded or distributed), and; *(ii)* at the interface between the control software and the SDN switches. SDN switches can indeed be seen as memory locations which are *read* and *modified* by various events and entities. While the first kind of interference can be detected with standard approaches [15], the second kind of interference is harder to detect as it often depends on a particular ordering of specific, but unpredictable events. Yet, detecting these interferences is important as they are typically at the root of deeper semantic problems such as blackholes, forwarding loops or non-deterministic forwarding.

This work In this paper, we present a system called SDNRacer, the first comprehensive dynamic and controller-agnostic concurrency analyzer for production-grade SDN controllers. SDNRacer checks for a variety of errors including (high-level) data races, packet coherence violations, and update isolation violations. It precisely captures the asyn-

chrony of SDN environments thanks to the first formulation of a *happens-before* (HB) model [24, 32] for the most commonly used OpenFlow features. Our HB relation is based on an in-depth study of the OpenFlow specification [1] and the behavior of network switches [2]. Further, we present a *commutativity specification* of an SDN switch under which two operations on the switch commute. This specification elegantly abstracts the behaviors of the switch and is a principled approach to reducing the number of false positives, enabling precise and scalable analysis. We illustrate the practicality of SDNRacer by analyzing real-world SDN controllers (both, single and multi-threaded) and show that it automatically discovers harmful and previously unknown concurrency errors.

Contributions The main contributions of this paper are:

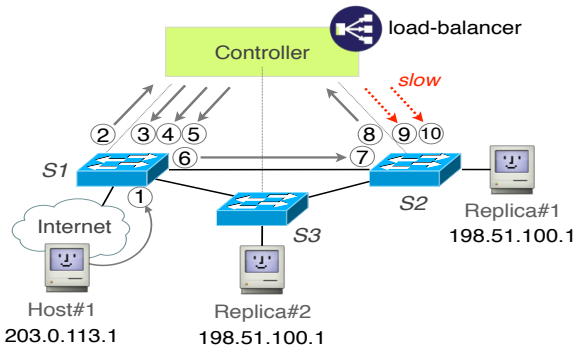
1. A thorough happens-before model which precisely captures the asynchronous interaction between an OpenFlow-based SDN controller and the SDN switches (§4).
2. A set of effective filters that dramatically reduce the number of reports, including a *commutativity specification* which captures the precise conditions under which two operations on the network switch commute (§5). Together, the specification and the filters reduce the number of reported issues by several orders of magnitude.
3. A complete implementation of SDNRacer, a dynamic analyzer which can readily analyze production-grade (single and multi-threaded) SDN controllers for various properties including: data races, per-packet consistency and update consistency (§7).
4. A comprehensive evaluation of SDNRacer attesting that it can uncover harmful and previously unknown bugs in existing SDN applications [7–11, 29–31, 35] (§8).

2. Overview

This section provides an overview of SDNRacer. We start with some background notions on concurrency issues in SDN programming (§2.1). Then, we give a motivating example illustrating how concurrency errors can arise and the negative effects they can have on the network (§2.2). We also explain how SDNRacer addresses the key problem of reducing the number of false positives (§2.3). Finally, we discuss how to use SDNRacer for detecting higher level properties (beyond races) such as consistency violations (§2.4).

2.1 SDN programming and concurrency issues

An SDN controller is an event-driven program whose goal is to compute, maintain and populate the forwarding table of each SDN switch in the network. The forwarding table of a switch is an ordered (by priority) list of forwarding entries composed, among other things, of a boolean predicate and a forwarding action. The predicate identifies a set of packets to which the corresponding forwarding action is applied.




```
if dst == server:
 _rep = rep[idx]
 idx = (idx+1)%2
 install_path(src,_rep)
 install_path(_rep, src)
 packet_out(pkt,in_sw)
...
```

```
def install_path(s,d):
 path = dijkstra(s,d)
 for i in path:
  flow_mod(i,s,d,
   fwd(p[i+1]))
```

Figure 1: An example of a simple load-balancing application (bottom) and a sequence of events (top), which leads to a forwarding loop. The cause of the issue is a concurrency error.

Forwarding actions include sending the packet to the controller or to a given output port.

SDN controllers operate in highly asynchronous environments where events such as packets arriving at a switch, link or node failures, or expiring flows can be dispatched to the controller at any time, all non-deterministically. Other events such as collecting various statistics from the switches are dispatched to the controller synchronously.

In the following, we say that a concurrency issue arises when there are two unordered accesses to the switch flow table, one of which is a write produced by the controller.

2.2 Example: a non-deterministic forwarding loop in a load balancer

Consider a simple controller program which runs a load-balancer application (see Fig. 1) that directs external requests to a chosen replica in a round-robin fashion.

Now, consider the following sequence of events: an external host, Host#1, sends a request directed to a farm of web server replicas identified by the IP address `198.51.100.1`. That request hits the first switch in the network, $S1$ ①; since it is a new request, $S1$ sends the request to the controller ② in an OpenFlow message called `PACKET_IN`. The controller *(i)* elects Replica#1; *(ii)* computes the shortest-path between $S1$ and Replica#1 (as well as the return path); *(iii)* pushes down two write events (called `FLOW_MODs`), one for each traffic direction, on $S1$ ③–④ and similarly on $S2$ ⑨–⑩ to forward the packets from Host 1 to Replica#1, and; *(iv)* sends the request back to $S1$ in a `PACKET_OUT` ⑤. $S1$ sends the request to $S2$ ⑥. In this trace, the packet hits $S2$ ⑦ *before* the corresponding flow rules ⑨–⑩ are installed on $S2$, causing the packet to be sent back to the controller ⑧. Assuming a round-robin selection algorithm, the controller now elects Replica#2, computes the shortest-path between

$S2$ and Replica#2 and pushes down the corresponding flow rules on $S2$, $S1$ and $S3$.

From this point on, the traffic is processed *incorrectly*, in a non-deterministic manner, as $S1$ and $S2$ each have forwarding entries with the same priority that match each direction of the traffic. Concretely, both directions of the traffic either end up caught in a forwarding loop, if $S1$ (resp. $S2$) uses the rule to forward the traffic to $S2$ (resp. $S3$), or hits one of the two replicas, non-deterministically. As replicas maintain state for each connection they receive, changing the replica on-the-fly will cause the connection to drop. In both cases, traffic ends up being lost.

In this example, the concurrency error arises between the *read event* caused by the packet received by $S2$ ⑦ and the *write event* ⑨ matching it which leads to lost traffic. Note that the controller could prevent this problem by using OpenFlow Barrier messages to ensure the rules are installed on both $S1$ and $S2$ before pushing the request back to $S1$.

Detecting such issues requires careful and precise definition of how ordering between operations is induced in the network as well as a definition for what it means for two events to interfere. We address both of these requirements in the paper by defining a happens-before model for SDNs as well as a commutativity specification of the flow table, enabling us to precisely state when two events interfere.

2.3 Reducing the amount of concurrency issues

A key problem that every practical concurrency analyzer must address is reducing the amount of reported issues that are false positives and therefore, harmless. SDNRacer filters numerous false positives by leveraging two distinct filters. Together, these two filters reduce the number of races by up to 99.97%. Detailed evaluation of the filtering performance of our tool is provided in §8.

Filter 1: Commuting events Commutativity relates to whether changing the order of two events affects the network state in different ways. If not, then even if two events are interfering with each other (via low level reads and writes), the network state ends up being identical. Such an interference is therefore harmless and can be filtered out.

Consider Fig. 1 again and the write events ③–④ that are pushed to $S1$ upon the reception by the controller of the packet sent by Host#1 ②. These two write events race with each other as the switch does not guarantee any ordering between write requests: either ③ will happen before or ④. However, the race is harmless as the two events are for non-overlapping entries of the forwarding table. In other words, the forwarding table at $S1$ will end up being identical independently of whether ③ happens before ④ or not. We say that ③ and ④ commute. Later in the paper, we present a precise formal definition of the commutativity specification of the forwarding table.

Filter 2: Time In theory, SDN switches can take an unbounded amount of time to perform a command (read or write). In practice though, they tend to execute them within a relatively short time frame. This observation enables SDNRacer to filter unlikely interference issues [19, 23, 38]. For instance, if a read and a write event are separated by, say 10 seconds, then they are unlikely to be reordered in practice. SDNRacer enables the SDN developer to specify a time δ after which two events cannot interfere anymore. This δ can easily be estimated based on the maximum network delay and the maximum switch processing time.

2.4 Detecting violations of high level properties

SDNRacer goes beyond detecting interferences and is capable of detecting violations of higher level properties such as inconsistent packet forwarding during a network update [37]. Update consistency means that packets are either forwarded by the old or the new version of the forwarding state, but not by an interleaving of the two.

So far, only a few SDN controllers such as Frenetic [17] guarantee update consistency. With SDNRacer, an SDN developer can now analyze *any* controller for consistency problems. In §8, we show that many such controllers (Floodlight [16], POX [28], ONOS [6]) are actually inconsistent. Most importantly, SDNRacer consistency analysis enabled us to discover previously unknown harmful bugs in several of them.

3. Formal Model of SDN operation

In this section, we define a formal model of a Software-Defined Network. This model includes both events occurring in the network as well as a model of the flow table in an OpenFlow switch. In later sections, we use this formalization to specify a precise happens-before (HB) relation and a commutativity specification of the flow table.

3.1 Operations and Events

We begin by defining a small set of events which succinctly encapsulate the relevant operations performed by the controller, the network switches, and hosts in the network. The operations are defined in §3.2.2 and contain the *reads* and *writes* (updates) to the flow table.

For each event type, we define a set of attributes that describe the event. Depending on the event type, only a subset of attributes is used: $\langle pid, mid, out\_pids, out\_mids, msg\_type, sw, ops\rangle$ where pid is the identifier of the packet processed by the event. Since network packets are potentially processed by more than one event, SDNRacer generates a Packet ID pid that does not map directly to any of the headers but rather it designates a specific packet in a specific event. mid is the identifier of the OpenFlow message processed by the event. If there are no such packets/messages, these attributes are set to the undefined value $\perp$. The set out_pids contains the identifiers of all packets emitted by the event. For each event that emits a packet (*e.g.*, SendPkt) SDNRacer will generate a new unique pid for the packet

and add it to its out_pids set. Each out_pids is a set because events emitting multiple packets will generate multiple new pids. For instance, SDN switches can duplicate packets and output them on multiple ports. The HB model uses the packet ids to link causally related events as defined in §4. The set out_mids contains the identifiers of all OpenFlow messages emitted by the event. Each out_mids is a set because the controller can issue multiple messages in response to one event. If there are no such packets or messages, these sets are empty $\varnothing$. For events where $mid \neq \perp$, the OpenFlow message processed by the event is of type msg_type. The relevant message types for our analysis are: `PACKET_IN`, `PACKET_OUT`, `BARRIER_REQUEST`, `BARRIER_REPLY`, `PORT_MOD`, `FLOW_REMOVED` and `FLOW_MOD`. Finally, sw is a switch identifier, and *ops* is the set of flow table operations the event contains.

The following events capture the behavior of the switches, controllers, and hosts:

HandlePkt(sw**,** pid**,** out_pids**,** out_mids**,** ops**)** denotes that a switch received and processed a data plane packet pid. There are three cases: *i)* either OpenFlow messages are generated, in which case out_mids contains the OpenFlow messages and out_pids contains the packet stored in the switch buffer; *ii)* a packet is forwarded, in which case out_pids contains the packet to be forwarded, or; *iii)* the packet is dropped.

HandleMsg(sw**,** mid**,** pid**,** out_pids**,** out_mids**,** msg_type**,** ops**)** denotes that the switch received and processed the OpenFlow message mid with type msg_type. The pid is $\perp$ unless a packet is read from the switch buffer. As a result of processing this packet, OpenFlow messages can be generated (in which case out_mids contains the OpenFlow messages), and a packet can be forwarded (in which case out_pids contains the packet to be forwarded).

SendPkt(sw**,** pid**,** out_pids**)** denotes that the switch sw sent the packet pid with a new identifier (in out_pids) out to another switch or host.

SendMsg(sw**,** mid**,** out_mids**)** denotes that a switch sent the OpenFlow message mid out to the controller with the identifier in out_mids.

RemovedFlow(sw**,** mid**,** out_mids**,** ops**)** denotes that a flow table entry in the switch timed out or was explicitly deleted. As a result of this event, a flow removed message may be generated (in which case the out_mids contains it).

CtrlHandleMsg(mid**,** out_mids**)** denotes that the controller received and processed the OpenFlow message mid, and generated the OpenFlow messages in out_mids in response.

CtrlSendMsg(mid**,** out_mids**)** denotes that the controller sent the OpenFlow message mid out to the control plane with the identifier in out_mids.

HostHandlePkt(pid**,** out_pids**)** denotes that a host received and processed the packet pid, and generated the packets in out_pids in response.

HostSendPkt(pid**,** out_pids**)** denotes that a host sw sent the packet pid with a new identifier (in out_pids) out to another switch or host.

3.2 A model of an SDN flow table

We now define a model of the flow table in an OpenFlow switch which contains a set of entries used to match packets.

3.2.1 Flow Table: Entries

A packet contains a *header* and a *payload*. The *header* consists of a set of fields (*e.g.*, IP source, IP destination or VLAN id) used to match packets against flow table entries. The *payload* is a sequence of bits and does not affect our specification (discussed later). For a packet pkt we use the notation $pkt.h$ to refer to the header associated with pkt.

Each flow table entry contains the fields *match*, *priority*, *actions*, *counters*, and *timeouts*. The *match* can be either an exact match or a wildcard match. *Priority* is a number specifying entry preference in case the packet matches multiple flow entries, and *actions* specify a set of forwarding operations to be performed on a matching packet. *Counters* contains values used for statistics, while *timeouts* contains hard and idle timeout values.

For a flow table entry e we use the shortcut notation $e.m$, $e.p$ and $e.a$ to refer to the *match*, *priority* and *actions*. A match between two entries e_1 and e_2 is exact, denoted as $e_1.m = e_2.m$, when all *match* fields are exactly the same (including the wildcards). A match between e_1 and e_2 is wildcard, denoted as $e_1.m \subseteq e_2.m$, if some of the fields in $e_1.m$ are not an exact match but contained in $e_2.m$ due to more permissive wildcards. The same definition of wildcard and exact match applies to a packet and to a flow table entry.

3.2.2 Flow Table: Operations

There are four types of operations that can be performed on the flow table: $read$ operations are part of $HandlePkt$ events and are performed for each received packet, while add, mod and del operations are part of $HandleMsg$ events with a msg_type of `FLOW_MOD` and are performed when the message is processed. In our work we used the OpenFlow specification 1.0 [1] to define the semantics of all of the above operations.

$read(pkt)/e_{read}$: The $read$ operation denotes that a packet pkt is matched against the flow table to determine the highest priority flow table entry e_{read} that should be applied. If there is no such flow table entry, e_{read} is set to the empty value $none$. Note that the value of e_{read} depends on the state of the flow table against which the packet pkt is being matched.

$add(e_{add}, no\_overlap)$: An add operation tries to add a new entry e_{add} to the flow table. If $no\_overlap$ is true then

SWITCHDATAPLANE:
$$\frac{\alpha \in HandlePkts \cup HandleMsgs \quad \beta \in SendPkts \quad \beta.pid \in \alpha.out\_pids}{\alpha \prec \beta}$$

SWITCHCONTROLPLANE:
$$\frac{\alpha \in HandlePkts \cup HandleMsgs \cup RemovedFlows \quad \beta \in SendMsgs \quad \beta.mid \in \alpha.out\_mids}{\alpha \prec \beta}$$

SWITCHBUFFER:
$$\frac{\alpha \in HandlePkts \cup HandleMsgs \quad \beta \in HandleMsgs \quad \beta.pid \in \alpha.out\_pids}{\alpha \prec \beta}$$

HOST:
$$\frac{\alpha \in HostHandlePkts \quad \beta \in HostSendPkts \quad \beta.pid \in \alpha.out\_pids}{\alpha \prec \beta}$$

CONTROLLER:
$$\frac{\alpha \in CtrlHandleMsgs \quad \beta \in CtrlSendMsgs \quad \beta.mid \in \alpha.out\_mids}{\alpha \prec \beta}$$

DATAPLANE:
$$\frac{\alpha \in SendPkts \cup HostSendPkts \quad \beta \in HandlePkts \cup HostHandlePkts \quad \beta.pid \in \alpha.out\_pids}{\alpha \prec \beta}$$

CONTROLPLANETO:
$$\frac{\alpha \in SendMsgs \quad \beta \in CtrlHandleMsgs \quad \beta.mid \in \alpha.out\_mids}{\alpha \prec \beta}$$

CONTROLPLANEFROM:
$$\frac{\alpha \in CtrlSendMsgs \quad \beta \in HandleMsgs \quad \beta.mid \in \alpha.out\_mids}{\alpha \prec \beta}$$

BARRIERPRE:
$$\frac{\alpha, \beta \in HandleMsgs \quad \alpha.msgtype = \texttt{BARRIER\_REQUEST} \quad \alpha.sw = \beta.sw \quad \alpha <_\pi \beta}{\alpha \prec \beta}$$

BARRIERPOST:
$$\frac{\alpha, \beta \in HandleMsgs \quad \beta.msgtype = \texttt{BARRIER\_REQUEST} \quad \alpha.sw = \beta.sw \quad \alpha <_\pi \beta}{\alpha \prec \beta}$$

TIME1:
$$\frac{\alpha \in HandlePkts \cup HandleMsgs \quad \beta \in HandleMsgs \quad \beta.t - \alpha.t > \delta}{\alpha \prec \beta}$$

TIME2:
$$\frac{\alpha \in HandleMsgs \quad \beta \in HandlePkts \cup HandleMsgs \quad \beta.t - \alpha.t > \delta}{\alpha \prec \beta}$$

Figure 2: Happens-before rules capturing ordering of packets and OpenFlow messages for a trace π.

a new entry is not added if a single packet may match both the new entry and an entry already in the flow table, and both entries have the same priority.

$mod(e_{mod}, strict)$: A mod operation modifies existing entries in the flow table. A boolean flag $strict$ is used to distinguish between the two types of modifications issued by the controller. In strict mode, an exact match (including the priorities) is used to determine whether an entry should be modified whereas in non-strict mode a wildcard match is used. Note that mod will act as an add in case no match is found.

$del(e_{del}, strict)$: A del operation deletes all entries that match the entry e_{del} in the flow table. Similarly to the mod operation, $strict$ affects how the matching is performed.

4. Happens-Before Model

In this section we define a precise happens-before (HB) model for SDNs (based on the events described earlier). To ensure correctness of the happens-before model, we designed the model based on an in-depth study of the OpenFlow switch specification [1] and the analysis of two software switch implementations: the POX software switch as well as the production quality Open vSwitch [2].

The HB relation is a binary relation $\prec \subseteq Event \times Event$ that is irreflexive and transitive. For convenience, we use the notation $\alpha \prec \beta$ instead of $(\alpha, \beta) \in \prec$. For a finite trace consisting of a sequence of events $\pi = \alpha_0 \cdot \alpha_1 \cdot \ldots \cdot \alpha_n$ we use $\alpha <_\pi \beta$ to denote that event α occurs before event β in π. We use $HandleMsgs$ to denote a set of all the events of type $HandleMsg$ and define such sets for each event type defined in §3. We illustrate the HB ordering rules induced from a given trace π in Fig. 2. All except four rules (BARRIERPRE, BARRIERPOST, TIME1, TIME2) make use of the information provided by the attributes pid, out_pids, mid, and out_pids. These capture the causality between two events α and β in the trace, where α caused β to happen. BARRIERPRE, BARRIERPOST describe the effect of `BARRIER_REQUEST` messages on OpenFlow switches. The rules TIME1 and TIME2 are speculative (discuss later).

We next proceed to describe our rules. We also illustrate the effect of each rule on the example shown in Fig. 1.

SWITCHDATAPLANE and SWITCHCONTROLPLANE: These rules order event processing packets and OpenFlow messages within a single switch. They order events that result in new $SendPkt$ and $SendMsg$ events before the new events. In our example, this rule introduces the orderings ① $\prec$ ②, ⑤ $\prec$ ⑥, and ⑦ $\prec$ ⑧.

SWITCHBUFFER: When sending a `PACKET_IN` message to the controller in a $SendMsg$ event, the full packet contents need not be contained inside the message. Instead, the switch may store the packet in its buffer and send only a part of the packet to the controller. Later, a $HandleMsg$ event of msg_type `PACKET_OUT` or `FLOW_MOD` may retrieve the packet from the buffer before processing it. This rule orders $HandlePkt$ and $HandleMsg$ events that store a packet in the switch buffer before the $HandleMsg$ event that eventually retrieves a packet from the switch's buffer. In the example, this rule introduces the ordering ① $\prec$ ⑤.

HOST: This rule orders the processing of the packet in a $HostHandlePkt$ event before the sending of the reply packets in $HostSendPkt$ events.

CONTROLLER: This rule orders the processing of the OpenFlow message in a $CtrlHandleMsg$ event before the sending of the reply messages in $CtrlSendMsg$ events. In the

example, this rule introduces the orderings ② ≺ ③, ② ≺ ④, ② ≺ ⑤, ② ≺ ⑨, and ② ≺ ⑩.

DATAPLANE: This rule orders events that send a packet before events that receive the packet. In the example, this rule introduces the ordering ⑥ ≺ ⑦.

CONTROLPLANETO and CONTROLPLANEFROM: These rules order events that send an OpenFlow message before events that receive the message. In our example, these rules order the send of ②, ③, ④, ⑤, ⑧, ⑨, and ⑩ before the respective receive.

BARRIER: For performance reasons, the switch is allowed to handle messages received from the controller in a different order from the one they were sent. To enforce ordering, the controller can issue a `BARRIER_REQUEST` message which ensures that the network switch finishes processing of all previously received messages (enforced by BARRIERPRE rule), before executing any messages beyond the `BARRIER_REQUEST` (enforced by BARRIERPOST rule). Note that the switch sends `BARRIER_REPLY` message to the controller once it finished processing `BARRIER_REQUEST` and all the messages before it.

SPECULATIVE TIME-BASED RULES This rule adds edges between events that are highly unlikely to be reordered due to the physical limits of the network. The value of δ depends on the specific parameters of the network. It should include the maximum delay that a packet might take traversing the network and the time window in which the OpenFlow switches can reorder write events. The proper value of δ can be inferred from related work that measured flow setup time in different environments and switches from various vendors [19, 23, 38]. We show the effect of choosing δ in §8.2.

5. Commutativity Specification

In this section we introduce a commutativity specification for an OpenFlow switch. This is an important component that has been used previously to improve concurrency of multicore systems [13] as well as to enhance the precision of program analyses dealing with interference [14] (here, it is important to reduce the number of reported false positives). As with the HB model, we designed the model based on an in-depth study of the OpenFlow switch specification [1] and experimental testing with Open vSwitch [2].

To define what commutativity means, we compare the results of two operations, in particular, flow table state and the returned values (if any) of the participating operations. We consider two flow tables to be in the same state if all their flow table entries contain identical *priority*, *match*, and *actions* fields. For the purposes of commutativity we ignore the *counters* and *timeout* fields as they are not used for matching packets or entries.

The commutativity specification is conveniently specified in a form of a logical predicate φ over pairs of operations. For a pair of operations a and b, the predicate φ_b^a evaluates to $true$ if operations commute and to $false$ otherwise.

Auxiliary Relations. We define three auxiliary functions. First, we overload the set intersection operator $e_1 \cap e_2$ for entry match structures $e.m$ (and packet headers $e.p$) and use it to compute all packet headers that may match both e_1 and e_2. Next, we use $e_1 \overset{strict}{\subseteq} e_2$ to model the semantics of table entry matching in regular and $strict$ modes as follows:

$$e_1 \overset{strict}{\subseteq} e_2 := \begin{cases} e_1.m = e_2.m \wedge e_1.p = e_2.p & \text{if } strict \\ e_1.m \subseteq e_2.m & \text{if } \neg strict \end{cases}$$

A $deletes$ predicate models the semantics of a delete operation and specifies whether an entry e can be deleted:

$$deletes(e_{del}, e, strict) := e \overset{strict}{\subseteq} e_{del} \wedge e.\texttt{out\_port} \subseteq e_{del}.\texttt{out\_port}$$

Commutativity Specification. The commutativity specification of an OpenFlow switch is shown in Fig. 3. All of the rules are written in the form that specifies when the operations do not commute which is then negated. We adopt this approach as the resulting rules are more intuitive to read. What follows is a description of some of the non-trivial rules.

$\varphi(add, add)$**:** Adding two entries does not commute if: (i) the second entry overwrites the first one, or (ii) the second entry is not added because the first entry is already in the table. The entries can overwrite each other only if both are added without the $no\_overlap$ option and their $match$ and $priority$ is identical. In this case the old entry is replaced with the new one and as long as their actions are different they do not commute. If at least one entry specifies the $no\_overlap$ option, then they do not commute if they have the same $priority$ and there exists an entry that can be matched by both entries.

$\varphi(add, mod)$**:** In case the $no\_overlap$ option is not set, add and mod do not commute in cases when they are allowed to modify the same entry with different actions. If $no\_overlap$ is set, then mod can add a new entry that overlaps with add which would result in add not being added.

$\varphi(del, mod)$**:** If mod affects only a single entry ($strict$ mode), we simply check whether this entry can be deleted. Otherwise, as long as both rules can match the same entry, they do not commute.

$\varphi(add, del)$**:** add and del do not commute if: (i) the added entry can be removed by a subsequent delete, or (ii) the delete does not remove the entry to be added but might enable adding it by removing some other entries. This situation arises when headers that may match add and del overlap.

$\varphi(mod, mod)$**:** If neither modify operation uses $strict$ mode then they do not commute if there is an entry that may match both. If they are both $strict$ then this entry needs to be exactly the same. Otherwise they do not commute if they are allowed to change the entry of each other.

$$
\begin{array}{lll}
\varphi^{read(pkt)/e_{read}}_{add(e_{add}, no\_overlap)} & := \begin{array}{l} \neg(e_{read} \neq none \wedge e_{read} = e_{add}) \\ \neg(pkt.h \subseteq e_{add}.m \wedge (e_{read} = none \\ \quad \vee (e_{read}.p \leq e_{add}.p \wedge e_{read}.a \neq e_{add}.a))) \end{array} & \begin{array}{l} \text{if } add <_\pi read \\ \text{if } read <_\pi add \end{array} \\
\varphi^{read(pkt)/e_{read}}_{mod(e_{mod}, strict)} & := \begin{array}{l} \neg(e_{read} \neq none \wedge e_{read} \overset{strict}{\subseteq} e_{mod} \wedge e_{read}.a = e_{mod}.a) \\ \neg(e_{read} \neq none \wedge pkt.h \subseteq e_{mod}.m \wedge e_{read}.a \neq e_{mod}.a) \end{array} & \begin{array}{l} \text{if } mod <_\pi read \\ \text{if } read <_\pi mod \end{array} \\
\varphi^{read(pkt)/e_{read}}_{del(e_{del}, strict)} & := \begin{array}{l} \neg(pkt.h \subseteq e_{del}.m) \\ \neg(e_{read} \neq none \wedge deletes(e_{del}, e_{read}, strict)) \end{array} & \begin{array}{l} \text{if } del <_\pi read \\ \text{if } read <_\pi del \end{array} \\
\varphi^{del(e_{del}, strict_{del})}_{mod(e_{mod}, strict_{mod})} & := \begin{array}{l} \neg(deletes(e_{del}, e_{mod}, true)) \\ \neg(e_{del}.m \cap e_{mod}.m \neq \emptyset) \end{array} & \begin{array}{l} \text{if } strict_{mod} \\ \text{otherwise} \end{array} \\
\varphi^{add(e_{add}, no\_overlap)}_{del(e_{del}, strict)} & := \neg(deletes(e_{del}, e_{add}, strict) \vee (no\_overlap \wedge e_{add} \cap e_{del} \neq \emptyset)) & \\
\varphi^{mod(e_1, strict_1)}_{mod(e_2, strict_2)} & := \begin{array}{l} \neg(e_1.m \cap e_2.m \neq \emptyset \wedge e_1.a \neq e_2.a) \\ \neg(e_1.m = e_2.m \wedge e_1.p = e_2.p \wedge e_1.a \neq e_2.a) \\ \neg((e_1 \overset{strict_2}{\subseteq} e_2 \vee e_2 \overset{strict_1}{\subseteq} e_1) \wedge e_1.a \neq e_2.a) \end{array} & \begin{array}{l} \text{if } \neg strict_1 \wedge \neg strict_2 \\ \text{if } strict_1 \wedge strict_2 \\ \text{otherwise} \end{array} \\
\varphi^{add(e_{add}, no\_overlap)}_{mod(e_{mod}, strict)} & := \begin{array}{l} \neg(e_{add} \overset{strict}{\subseteq} e_{mod} \wedge e_{add}.a \neq e_{mod}.a) \\ \neg(e_{add} \cap e_{mod} \neq \emptyset) \end{array} & \begin{array}{l} \text{if } \neg no\_overlap \\ \text{otherwise} \end{array} \\
\varphi^{add(e_1, no\_overlap_1)}_{add(e_2, no\_overlap_2)} & := \begin{array}{l} \neg(e_1.m \cap e_2.m \neq \emptyset \wedge e_1.p = e_2.p) \\ \neg(e_1.m = e_2.m \wedge e_1.p = e_2.p \wedge e_1.a \neq e_2.a) \end{array} & \begin{array}{l} \text{if } no\_overlap_1 \vee no\_overlap_2 \\ \text{otherwise} \end{array}
\end{array}
$$

Figure 3: Commutativity specification of an OpenFlow switch. Two $read$ or two del operations always commute.

$\varphi(read, add/mod/del)$**:** For $read$ operations we distinguish two cases depending on the order in which the operations are executed in the trace. If a $read$ happens first, the operations do not commute if the matched entry is not guaranteed to match after the second operation is performed. Since we know the concrete flow entry that matched the initial read, such a check can be performed precisely. In the case of a $read$ executing second, we simply check whether the matched rule is identical to the one added or modified. For a del operation, we conservatively check whether an entry that matches the packet can be removed.

Key Points. Note, that for the $read$ operations our commutativity specification incorporates parts of the flow table state by using the returned values. Further, commutativity rules for $read$ are specialized based on the trace order, which is a direct consequence of depending on the state in which the operations were performed. However, commutativity checking remains efficient, as no flow table state beyond these return values needs to be stored or simulated.

6. Consistency Properties

In this section, we discuss the checking of two important previously defined consistency related properties (§6.2 and §6.3). A useful guarantee of our checking approach is that if we establish the properties holding on a single trace, it follows that the properties hold for all traces which contain the same events (though perhaps events appear in a different order) where the traces use the same input (§6.4). This guarantee reduces the number of traces we need to explore per input.

6.1 Network Update

SDN applications typically update more than one flow rule in the network to reflect entire policy changes; *e.g.* re-routing congested traffic through a different end-to-end path. To capture this behavior, we map individual events containing write operations in a trace π into sets of network updates, such that each set Γ of network updates reflects a policy change in the network. Network updates are either triggered reactively by messages from switches (*e.g.*, `PACKET_IN` messages), or proactively by an external event (e.g., manual change from a network operator).

In reactive applications such as a learning switch, we can use the happens-before model to extract the set Γ of events that are part of a reactive update for the event α. For reactive updates, α is of type $RemovedFlow$ or $SendMsg$. More formally, a reactive update for event α is the set of events defined as follows:

$$
\begin{array}{l}
R(\pi, \alpha) ::= \\
\{\beta \mid \beta \in \pi \wedge \alpha \prec \gamma_1 \prec \gamma_2 \prec \beta \\
\quad \wedge\ \gamma_1 \in CtrlHandleMsgs \quad \wedge\ \gamma_1 \in Succ(\alpha) \\
\quad \wedge\ \gamma_2 \in CtrlSendMsgs \quad \wedge\ \gamma_2 \in Succ(\gamma_1) \\
\qquad\qquad\qquad\qquad \wedge\ \beta \in Succ(\gamma_2)\}
\end{array}
$$

where $Succ(\gamma)$ returns all events created directly as a result of processing event γ:

$$Succ(\gamma) ::= \{e \mid e.pid \in \gamma.pid\_outs\}$$

On the other hand, in proactive applications, such as a static flow pusher, the updates are caused by external events or internal controller configurations and hence are outside the scope of our HB model. We provide two options

to group proactive write events into network updates. The controller can annotate the writes with the version number. Alternatively, to keep controller instrumentation to a minimum, we provide a heuristic to detect proactive updates. The heuristic uses a clustering algorithm to group events together based on time into a set Γ of network updates. Then, we merge different clusters if there is a barrier request in one cluster and the response in another. This merge operation mitigates clustering errors from slow network updates.

Commutativity race Beyond standard read-write data races, a core high level property that we check is commutativity races [14]. A commutativity race occurs when two events: (i) do not commute according to our commutativity specification, and (ii) the events are unordered by our happens-before relation. Given a trace π, we denote the set of commutativity races in π as $CR(\pi)$.

Further, for the reported commutativity races the same guarantees as in existing state-of-the-art commutativity happens-before race detectors [14] are provided. In particular: (i) the first reported race is always guaranteed to be a real race, and (ii) if no race is reported for the given execution, then no execution from the same input state contains a race.

6.2 Update Isolation

Wang *et al.* [3] define a set of policy changes to be *isolated* if they do not interfere with each other. That is, executing the updates defined by each policy in any interleaving results in a network state that is equivalent to one that is obtained by some serial execution. We check if a set of multiple policy changes $\Gamma^\star = \{\Gamma_1, \Gamma_2, \Gamma_3, \ldots\}$ is isolated, by checking if no pair of events (across different policy changes) is in the set of commutativity races $CR(\pi)$:

$$\begin{aligned} UI(\Gamma^\star) ::= \quad & \nexists\, \alpha, \beta : (\alpha, \beta) \in CR(\pi) \wedge \\ & \alpha \in \Gamma_u \wedge \beta \in \Gamma_v \wedge \Gamma_u \neq \Gamma_v \end{aligned}$$

6.3 Packet Coherence

The next property we check is coherence of a packet trace. We say that a packet trace is coherent if each packet is processed entirely using one consistent global network configuration [25, 37]. To check for this property, given a trace π, we first define the notion of a packet trace which is a subset of events that participate in processing packet pkt as it traverses throughout the network until the packet reaches a destination host. An event trace $\tau(\pi, \gamma)$ is a subset of the events in trace π that were created as a result of processing event γ. We say that event trace $\tau(\pi, \gamma)$ corresponds to a packet trace for a given packet pkt if event γ originated the packet pkt. More formally, the event trace $\tau(\pi, \gamma)$ is defined as follows:

$$\begin{aligned} \tau(\pi, \gamma) ::= \quad & \gamma \cup \{\tau(\pi, \beta) \mid \quad \beta \in Succ(\gamma) \wedge \\ & \beta \notin HostHandlePkts\} \end{aligned}$$

Then, we write $CR_\tau(\pi, \gamma)$ to denote all races where one of the racing events is in $\tau(\pi, \gamma)$.

$$CR_\tau(\pi, \gamma) ::= \{(\alpha, \beta) \mid \alpha \in \tau(\pi, \gamma) \ \wedge \ (\alpha, \beta) \in CR(\pi)\}$$

We check packet coherence for all packets pkt in a given trace π, *i.e*, we check coherence for each packet trace $\tau(\pi, \gamma)$ extracted from the trace π. We can be certain that a packet trace $\tau(\pi, \gamma)$ exhibits packet coherence if $CR_\tau(\pi, \gamma) = \varnothing$: any network update that could affect the packet trace would introduce at least one race between the previous network state and the updated state.

However, under certain conditions a packet trace can be coherent in the presence of races, *i.e.*, when $CR_\tau(\pi, \gamma) \neq \varnothing$. Then, there is packet coherence if *(i)* there is only a single event e (containing flow table read operations on a single switch sw) that is part of any races in $CR_\tau(\pi, \gamma)$, and *(ii)* there are no events in $CR_\tau(\pi, \gamma)$ that modify any switches other than sw.

$$\begin{aligned} PC(\pi, \gamma) ::= \quad & CR_\tau(\pi, \gamma) = \varnothing \\ & \vee (\exists\, e : (\forall (\alpha, \beta) \in CR_\tau(\pi, \gamma) : \alpha = e \\ & \wedge \forall (\alpha, \beta) \in CR_\tau(\pi, \gamma) : \beta.sw = e.sw)) \end{aligned}$$

Intuitively, this means that there can be packet coherence even in the presence of races, if the reordering of the single event in the races does not negatively affect packet coherence. This is possible if there is only a single such event, *i.e*, if there are only two possible reorderings.

6.4 Guarantees

We note that our checks for the properties discussed above are more general than simply taking snapshots of the flow tables [20–22], as verification of a static snapshot does not consider event reorderings. Even though a trace π may be free of violations, there may be another trace π' with the same inputs as π which does contain violations. In contrast, our checks on π guarantee that any such trace π' is free of violations, which is useful as it means we do not need to explore all possible traces π'. Our guarantee is standard in happens-before classic race detectors [14, 15], however, here we ensure the guarantee even beyond races.

7. Implementation

We implemented a full prototype of SDNRacer in around $3,000$ lines of Python code[1]. The implementation consists of three parts: (i) an instrumentation of the SDN troubleshooting system STS [39]; (ii) an instrumentation of several controller frameworks (POX, Floodlight, ONOS), and; (iii) a concurrency analyzer that implements the happens-before rules, commutativity checks, and consistency checks.

Network instrumentation STS simulates a complete network, including OpenFlow switches, links, and hosts. We instrumented STS to further track packets, messages and switch operations and write them to a file.

(Optional) controller instrumentation The controller instrumentation for POX, Floodlight, and ONOS includes a wrapper around the respective event handlers for incoming

[1] https://github.com/nsg-ethz/SDNRacer

Controller	POX	FloodLight	ONOS
LoC	40	139	55

Table 1: While SDNRacer *does not* require controller instrumentation, few lines of instrumentation code enables to filter harmless issues (around 20% more).

messages, and links the incoming message with the corresponding outgoing message, when possible. Instrumenting the controller only requires few lines of code (Table 1). The controller instrumentation then passes this information to STS. Instrumenting the controller is *not needed* for SDNRacer to work, but it helps in filtering harmless concurrency issues by adding more HB orderings in addition to those defined in §4 (*e.g.*, from 314 to 239 reported races, 23.9%, in one experiment). POX uses cooperative threading and runs only one task at any given time while Floodlight and ONOS are multi-threaded and they context-switch threads. However, this is not relevant to our model because SDNRacer treats the controller as a blackbox, allowing us to use SDNRacer on a wide set of controllers with minimal instrumentation in the controller framework. A more specific approach would allow for more precision at the price of being controller-specific.

SDNRacer SDNRacer reads events from a trace file, builds the HB graph and then runs the concurrency analysis on top of it. The HB graph as well as the races and inconsistent packets are output graphically for further inspection.

8. Evaluation

In this section, we evaluate SDNRacer's performance and usability. After describing our setup (§8.1), we first show that SDNRacer detects many consistency issues in existing controllers. As the number of issues is often large, we also show that SDNRacer can efficiently reduce the number of reported issues through filtering (§8.2). Second, we show several examples of consistency violations discovered by SDNRacer (§8.3). Finally, we show that SDNRacer is fast and completes its analysis in few seconds on large traces containing thousands of events (§8.4). Our results indicate that SDNRacer is an effective tool for troubleshooting real-world SDN deployments.

8.1 Experimental Setup

We ran SDNRacer on a set of network traces collected from a representative set of SDN controllers, running different existing applications, on different network topologies. All experiments were performed on a machine with 16GB of RAM and a modern 4-core processor running at 2.5GHz.

SDN controllers We run SDNRacer against three controllers: Floodlight version 0.91 [16], POX EEL [28], and ONOS version 1.2.2 [6]. We further instrumented them to better track HB relationships (Table 1).

Applications We choose 5 representative applications including purely proactive and pure reactive applications. Unless specified otherwise, we run the same application on each controller. The implementation of all analyzed applications is included as part of the official controller distribution.

App#1. MAC-learning: A purely reactive application builds and maintains a dynamic MAC address table for each switch. This table maps known MAC addresses to the physical port on which they can be reached. We analyze the implementations shipped with Floodlight and POX [10, 31].

App#2. Forwarding: MAC-learning applications are highly inefficient as they work on a per-switch basis. To alleviate this, most controllers include a "Forwarding Application" which works at the network-level and reactively builds and maintains one network-wide MAC address table. We analyze the implementations shipped with Floodlight, POX, ONOS [9, 29, 30, 35].

App#3. Circuit Pusher: This purely proactive application automatically installs paths between two hosts identified by their MAC addresses, as well as the switch and port they are connected to. We analyze the implementation shipped with Floodlight [7].

App#4. Admission Control: This application allows/drops host communication based on given operator policies. We analyze the implementation shipped with Floodlight [8].

App#5. Load Balancer: This application performs stateless load balancing among a set of replica identified by a virtual IP address (VIP). Upon receiving packets destined to a VIP, the application selects a particular host and installs flow rules along the entire path. We analyze the implementation shipped with Floodlight [11].

Topologies We ran each controller on three different topologies: *Single*, *Linear*, and *BinTree*. *Single* has one switch with two hosts. *Linear* has two switches with one host connected to each. *BinTree* has seven switches connected as a binary tree with four hosts connected to leaf switches.

Traces We collected 29 traces using STS and a mix of applications, controllers, and network topologies. The traces have between 193 and 24,612 events spanning between 26 and 74 seconds (Table 2). Each trace is the result of 200 STS simulation steps. In every step, each host in the topology decides randomly whether it is going send a packet to another randomly chosen host.

Some applications required additional parameters to run. For Circuit Pusher, we install a new circuit every second between two randomly selected hosts as well as remove one existing circuit with a probability of 0.5. For Admission Control, we allow 80% of the hosts (randomly selected) to communicate. For Load Balancer, we create replica pools with two hosts and assign them a VIP. All hosts send traffic to the VIP. Since Load Balancer only makes sense with more than two hosts, we run it on larger topologies: (*Single4* and *Linear4*), connecting four hosts instead of two.

App	Topology	Controller	Events			Races				Updates		Packet Coherence		
			Events	WR	RD	Races	Comm.	Time	Remain.	Num	¬ Isolt.	Pkts	Racing	Incoh
LearningSwitch	Single	POX EEL	193	7	42	294	218	66	**10 (3.40%)**	6	0	42	10	**0**
		Floodlight	314	7	70	494	223	227	**44 (8.91%)**	5	0	70	18	**0**
	Linear	POX EEL	274	16	66	532	387	121	**24 (4.51%)**	18	0	34	11	**5**
		Floodlight	233	6	66	190	64	125	**1 (0.53%)**	5	0	33	1	**0**
	BinTree	POX EEL	4033	487	663	62066	61337	664	**65 (0.10%)**	402	0	190	28	**18**
		Floodlight	9320	1251	904	275257	270217	4737	**302 (0.11%)**	1156	34	223	119	**72**
Forwarding	Single	POX Angler	106	4	16	61	33	21	**7 (11.48%)**	12	0	16	7	**0**
		POX EEL	145	8	19	109	84	24	**1 (0.92%)**	12	0	17	1	**0**
		POX EEL Fx	184	8	29	189	145	42	**1 (0.53%)**	12	0	26	2	**1**
		ONOS	476	18	71	1336	1163	127	**14 (1.05%)**	3	0	73	27	**22**
		Floodlight	97	3	13	35	13	14	**8 (22.86%)**	5	0	13	8	**0**
	Linear	POX Angler	248	13	48	323	116	191	**12 (3.72%)**	31	1	20	6	**6**
		POX EEL	306	20	50	405	235	159	**7 (1.73%)**	28	1	20	7	**7**
		POX EEL Fx	303	16	51	276	206	62	**4 (1.45%)**	27	0	20	3	**3**
		ONOS	880	44	181	4059	3781	228	**49 (1.21%)**	11	0	76	19	**15**
		Floodlight	180	6	36	104	46	45	**13 (12.50%)**	5	0	14	5	**5**
	BinTree	POX Angler	2106	286	359	20447	13179	6988	**272 (1.33%)**	127	4	77	43	**42**
		POX EEL	4362	504	453	34385	27956	6201	**219 (0.64%)**	138	3	86	59	**58**
		POX EEL Fx	4283	467	413	12509	12238	242	**24 (0.19%)**	147	0	92	61	**55**
		ONOS	8031	1492	920	236429	233578	2598	**239 (0.10%)**	37	0	131	66	**49**
		Floodlight	1886	203	323	12293	11766	317	**209 (1.70%)**	71	0	76	57	**53**
CircuitPusher	Single	Floodlight	218	25	41	1301	1040	218	**43 (3.31%)**	8	1	41	35	**0**
	Linear	Floodlight	327	42	74	1933	1581	287	**65 (3.36%)**	10	1	38	34	**21**
	BinTree	Floodlight	1200	144	227	6156	5605	507	**44 (0.71%)**	14	3	142	10	**6**
Adm. Ctrl.	Single	Floodlight	190	3	36	104	35	62	**6 (5.77%)**	5	0	36	7	**0**
	Linear	Floodlight	221	6	48	139	56	69	**14 (10.07%)**	6	0	21	6	**6**
	BinTree	Floodlight	841	52	170	1384	1090	228	**66 (4.77%)**	25	0	74	20	**10**
LoadBalancer	Single4	Floodlight	3889	822	476	703864	685158	16492	**2214 (0.31%)**	449	1114	77	22	**0**
	BinTree	Floodlight	24612	6213	2163	4705379	4642118	62031	**1230 (0.03%)**	1419	464	226	101	**97**

Table 2: Reported races and properties violations for different traces with applying time filter using $\delta = 2$. The numbers in bold are the final numbers of races and incoherent packets reported to the user of SDNRacer.

8.2 Race Detection and filtering efficiency

SDNRacer reports many races (Table 2) whose actual number depends on the number of read and write events which in turn depend on the controller running the application. As an illustration, the same set of inputs led to 16 reads and 66 writes for the MAC-learning application running on POX EEL, but only six reads and 66 writes when running on Floodlight.

Reporting too many races is of little use to the developer. So, to be of practical use, SDNRacer is equipped with a set of filters based on commuting events, timing and race coverage [36]. We now evaluate the efficiency of each filter in turn. When all filters are applied, SDNRacer manages to filter out more than 90% of the races in the vast majority of cases.

Filter 1. Commutativity Commutativity is a major contributor to reducing the number of reported races. This filter alone reduces at least 33% of the races in almost all traces and more than 73% of races in 65.5% of the traces (Fig. 4).

Commutativity filtering performs best in traces that have many unrelated reads and writes. This high number of disjoint reads and writes is often the result of different hosts sharing the same path. For example, 91% of all races reported by running Circuit Pusher on the BinTree topology commute (Table 2) as the events relate to different hosts and non-overlapping entries.

Filter 2. Time-based Time filtering further helps reduce more than 20% of the races in about half of our traces (Fig. 5) with a δ value of 2 seconds (§4).

In Fig. 5, we report how much filtering is done as a function of δ. If δ is set to a high value, more false-positive races will be reported. For instance, if δ is set to 8 seconds, the time filter can only reduce up to 34.5% of the races in its best case. In contrast, it can filter up to 51.7% of the races in its best case when δ is set to 2 seconds. For our evaluation, 2 seconds is safe given our switch implementation and network size.

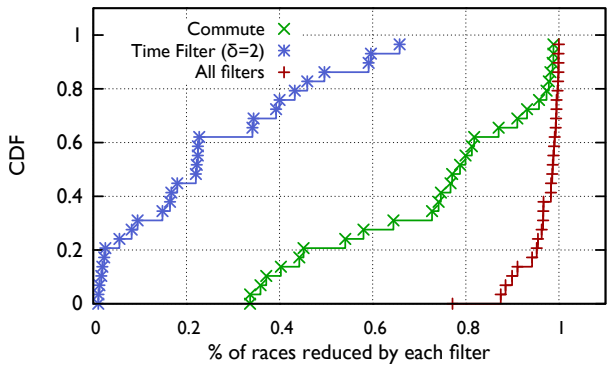


Figure 4: *The effect of SDNRacer filters.* When all filters are applied, more than 90% of all races are filtered in 89% of the cases. Commutativity filtering is the most efficient, followed by Time-based filtering.

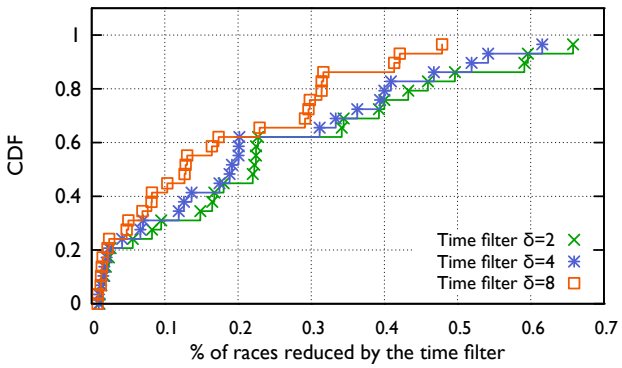


Figure 5: *The effect of time-based filter.* Choosing smaller δ filters more races. With δ=2, SDNRacer can filter more than 20% of the races in 48% of the cases. While with δ=8, SDNRacer can filter 20% of the races in 40% of the cases.

Other filters Like all happens-before detectors (*e.g.*, FastTrack [15]), SDNRacer's checks are as precise as the happens-before model. Hence, there can be false positives for covered races [36] due to data dependencies. To discover such cases, in addition to commutativity-based and time-based filtering, SDNRacer provides an additional filter that discovers covered races. Covered races are reported interferences that cannot happen because of high-level dependencies. We observed however that covered races account for only up to 2.4% of the races. As such, to speed-up processing, SDNRacer does not enable that filter by default.

8.3 Consistency Checks

SDNRacer detected consistency violations in all applications and controllers used in our experiments. In many cases, these violations turned out to be subtle (and some are unknown) bugs. In the following, we detail both update isolation and packet coherence violations (§6). Recall that the former leads to ultimately different network states being installed in the network while the latter relates to packets being forwarded according to different policies.

Violations of update isolation SDNRacer discovered update consistency violations in four applications (10 out of 29 traces): MAC-Learning, Forwarding, Circuit Pusher, and Load Balancer. For the Load Balancer application, the violation was the source of a serious bug.

- *Violation#1: Floodlight Load Balancer distributes flows inconsistently*. SDNRacer reports 1114 inconsistent network updates on the single switch topology and 464 inconsistent network update on the BinTree topology (Table 2).

  At a first glance, the number of update isolation violations might seem high. However, the vast majority of the violations are symptoms of the same bug. By analyzing the reported violations by SDNRacer and the application code we realized that, upon packet reception, the controller selects a replica, pushes flow rules to direct traffic to it, sends the packet back in the network towards the replica *without* waiting for the flow rule to be committed to the switch. As the rules are being installed, further packets go to the controller and trigger the process again. Concretely, this means that multiple load-balancing decisions can be taken for the same flow. Inconsistent flow assignments can lead to bad performance but also to connection drops as the same flow can be assigned to different replicas.

  Fix: The bug is easily fixed by forcing the Load-Balancer to request a barrier before pushing packets back into the network and by having it buffer (or drop) any subsequent packets it receives for the same connection.

- *Violation#2: POX forwarding module deletes rules installed by other modules*. SDNRacer reports an inconsistent update where a removal of flow induced by one module raced with a flow insertion induced by another module. Investigating the application code, we found out that the rules installed by the Discovery module (in charge of learning the network topology) were deleted by the Forwarding module whenever the topology changed.

  In this specific case the race between the two modules was not harmful as the default action directs packets to the controller anyway. This ensured that even though rules from the Discovery modules were deleted, it was still able to learn the topology. We stress that in newer versions of OpenFlow, the default action is now to drop packets, meaning this bug would cause the entire network traffic to be dropped whenever the topology changes.

  Fix. This bug is easily fixed by ensuring that the Forwarding application only deletes its own flow rules.

Violations of packet coherence SDNRacer discovered per-packet coherence violations in almost all traces (Table 2). Most of the incoherent packet cases concerned races occurring when the controller installs a set of flow rules and then sends a packet matching these flow rules without waiting for

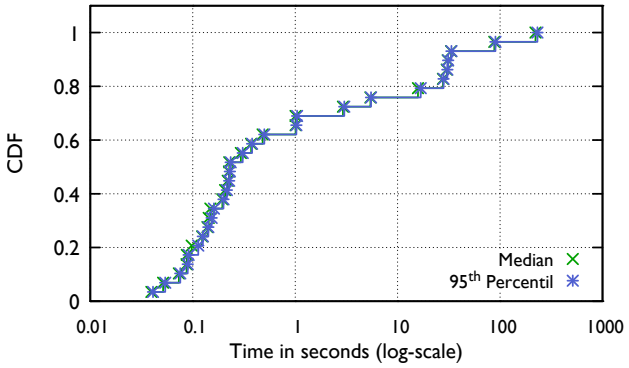


Figure 6: Analysis time for traces from Table 2. In 90% of the cases SDNRacer can analyze the traces in less than 30 seconds.

the flow rules to be committed first. As such, these type of races occurred more often in traces of reactive applications such as the Forwarding application. While waiting for writes to be committed is an obvious solution, it also slows down network operations indicating that many controllers trade consistency for speed. In general, violating per-packet coherence may not always be harmful. Poorly performed policy updates, for instance, can create per-packet coherence violations without leading to data losses. Even in this case, we believe that it is still important to report and quantify violations of per-packet coherence as correctness predicated on policy content is undesirable.

8.4 Time

SDNRacer finishes its analysis in less than 32 seconds in the vast majority of traces (Fig. 6). To measure this, we ran SDNRacer 20 times and collected the total time for: *(i)* loading the trace; *(ii)* building the HB graph; *(iii)* applying all filters, and; *(iv)* performing all consistency analysis.

The worst case (3.7 minutes) happened when SDNRacer analyzed the FloodLight Load Balancer on the BinTree topology. This long running time is due to a bug in the application (see §8.3) that caused the trace to have an order of magnitude more events and races than other traces.

9. Related Work

Data plane verification Several projects are aimed at verifying the correctness of SDNs. Anteater [26], HSA [21] and Libra [40] collect snapshots of the network forwarding state and check if it violates certain properties. VeriFlow [22] and NetPlumber [20] build on this by allowing real-time checking upon network updates. An extension [5] of VeriFlow allows using assertions to check network properties during controller execution. Similar to SDNRacer, these tools can detect interesting invariant violations. However, they cannot tell what precise sequence of events led to them, only that the latest update triggered the violation. STS [39] extends these works by considering the minimal sequence of events responsible for a given invariant violation. Unlike SDNRacer, STS does not have a precise formal specification of the partial orderings of events or the conditions under which two operations commute. As a result, STS cannot detect bugs unless the invariant is violated in a given trace. On the other hand, SDNRacer reports strictly more violations than STS by generalizing the observed trace to all traces obtainable from the same inputs. Additionally, STS uses network-wide snapshots to check various properties while SDNRacer considers all relevant events and thus does not miss any harmful violations. Finally, the output of SDNRacer and STS is different. STS outputs the minimal sequence of input events that reproduce an invariant violation while SDNRacer outputs the exact pairs of read/write events that caused the property violation.

Controller verification Other approaches seek to eliminate controller bugs, for instance, by synthesizing provably correct controllers [18]. Similarly, in FlowLog [34], rulesets are partially compiled to NetCore [33] policies and then verified.

NICE [12] uses concolic execution of Python controller programs with symbolic packets and then runs a model checker to determine invariant violations. Kuai [27] uses a simplified version of an OpenFlow switch as well as a custom controller language, but applies partial order reduction techniques to reduce the number of states the model checker has to explore. Although significantly more performant, Kuai still suffers from the state-space explosion problem associated with full model checking. Vericon [4] converts programs into first-order logic formulas and uses a theorem prover to verify safety properties. In contrast, SDNRacer is a dynamic analyzer that operates on actual controller traces and can quickly detect concurrency issues: the root cause of many bugs. The speed of the analysis only depends on the trace size, not on the controller. Previous approaches could benefit from our formal specifications in order to speed-up their verification time, *e.g.*, by not checking operations that do not interfere with the network state.

10. Conclusion

In this paper, we presented SDNRacer, the first scalable analysis system for finding a variety of concurrency-induced errors including (high-level) data races, per-packet consistency, and update consistency. SDNRacer makes several key contributions: *(i)* a precise formal happens-before model of SDN (OpenFlow) concurrency; *(ii)* efficient filters including a commutativity specification of a network switch, and; *(iii)* a thorough experimental evaluation illustrating that our techniques for filtering races and identifying high-level (consistency) violations work in practice. SDNRacer was also able to identify previously unknown and harmful bugs in existing SDN controllers.

Acknowledgements We thank the anonymous reviewers and our shepherd, Ben Liblit, for their insightful feedback.

References

[1] OpenFlow Switch Specification. Version 1.0.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

[2] Open vSwitch. Production Quality, Multilayer Open Virtual Switch. http://openvswitch.org/.

[3] Anduo, W. Zhou, B. Godfrey, and M. Caesar. Software-Defined Networks as Databases. In *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA, 2014. USENIX. URL https://www.usenix.org/conference/ons2014/technical-sessions/presentation/wang.

[4] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In ACM PLDI '14. doi: 10.1145/2594291.2594317.

[5] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An Assertion Language for Debugging SDN Applications. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14. ACM. doi: 10.1145/2620728.2620743.

[6] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14. ACM. doi: 10.1145/2620728.2620744.

[7] Big Switch Networks, Inc. Floodlight Circuit Pusher Application. https://github.com/floodlight/floodlight/tree/v0.91/apps/circuitpusher, 2013.

[8] Big Switch Networks, Inc. Floodlight Firewall. https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/firewall, 2013.

[9] Big Switch Networks, Inc. Floodlight Forwarding Application. https://github.com/floodlight/floodlight/blob/v0.91/src/main/java/net/floodlightcontroller/forwarding/Forwarding.java, 2013.

[10] Big Switch Networks, Inc. Floodlight Learning Switch. https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/learningswitch, 2013.

[11] Big Switch Networks, Inc. Floodlight Load-Balancer Application. https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/loadbalancer, 2013.

[12] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX NSDI '12*.

[13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In ACM SOSP '13. doi: 10.1145/2517349.2522712.

[14] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity Race Detection. In ACM PLDI '14. doi: 10.1145/2594291.2594322.

[15] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In ACM PLDI '09. doi: 10.1145/1542476.1542490.

[16] Floodlight. Floodlight Open SDN Controller. http://projectfloodlight.org/floodlight.

[17] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ACM ICFP '11*. doi: 10.1145/2034773.2034812.

[18] A. Guha, M. Reitblatt, and N. Foster. Machine-verified Network Controllers. In ACM PLDI '13. doi: 10.1145/2491956.2462178.

[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *ACM SIGCOMM '13*. doi: 10.1145/2486001.2486019.

[20] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *USENIX NSDI '13*, .

[21] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI '12*, .

[22] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying Network-wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.*, 42(4), Sept. 2012. doi: 10.1145/2377677.2377766.

[23] M. Kuźniar, P. Perešíni, and D. Kostić. What You Need to Know About SDN Flow Tables. In *International Conference on Passive and Active Measurement*, PAM '15. Springer International Publishing. doi: 10.1007/978-3-319-15509-8_26.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978. doi: 10.1145/359545.359563.

[25] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, 2013. doi: 10.1145/2535771.2535791.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM '11*. doi: 10.1145/2018436.2018470.

[27] R. Majumdar, S. D. Tetali, and Z. Wang. Kuai: A Model Checker for Software-defined Networks. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14.

[28] J. Mccauley. POX: A Python-based OpenFlow Controller. https://github.com/noxrepo/pox.

[29] J. McCauley. POX Angler Forwarding Application. https://github.com/noxrepo/pox/blob/angler/pox/forwarding/l2_multi.py, 2012.

[30] J. McCauley. POX EEL Forwarding Application. https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_multi.py, 2015.

[31] J. McCauley. POX EEL L2 Learning Switch. https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_learning.py, 2015.

[32] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. SDNRacer: Detecting Concurrency Violations in Software-defined Networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, in ACM SOSR '15. doi: 10.1145/2774993.2775004.

[33] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *ACM POPL '12*. doi: 10.1145/2103656.2103685.

[34] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-defined Networks. In *USENIX NSDI '14*.

[35] Open Networking Laboratory. ONOS (Open Network Operating System): Forwarding Application. https://github.com/opennetworkinglab/onos/tree/onos-1.2/apps/fwd, 2015.

[36] V. Raychev, M. Vechev, and M. Sridharan. Effective Race Detection for Event-driven Programs. In *ACM OOPSLA '13*. doi: 10.1145/2509136.2509538.

[37] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *ACM SIGCOMM '12*. doi: 10.1145/2342356.2342427.

[38] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *International Conference on Passive and Active Measurement*, PAM'12. Springer-Verlag. doi: 10.1007/978-3-642-28537-0_9.

[39] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *ACM SIGCOMM '14*. doi: 10.1145/2619239.2626304.

[40] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *USENIX NSDI '14*.