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.2.5 [Software Engineering]: Testing and Debugging; D.2.4 [Software Engineering]: Software/Program Verification

Keywords 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 asynchrony 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.

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

**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) selects 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 ⃝2→9 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 ⃝8. In this trace, the packet hits S2 ⃝7 before the corresponding flow rules ⃝8→9 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

### Diagram

![Diagram of load-balancing](image-url)
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 and the write event that 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. 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: \( (\text{pid, mid, out}_p, \text{out}_m, \text{msg}_t, \text{sw}, \text{ops}) \) 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 ⊥. 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 ∅. For events where mid ≠ ⊥, 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 ⊥ 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 e1 and e2 is exact, denoted as e1.m = e2.m, when all match fields are exactly the same (including the wildcards). A match between e1 and e2 is wildcard, denoted as e1.m ⊆ e2.m, if some of the fields in e1,m are not an exact match but contained in e2.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: A 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
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.

\( \text{mod}(e_{\text{mod}}, \text{strict}) \): A \text{mod} operation modifies existing
entries in the flow table. A boolean flag \text{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 \text{mod} will act as an \text{add} in case no match is
found.

\( \text{del}(e_{\text{del}}, \text{strict}) \): A \text{del} operation deletes all entries that
match the entry \( e_{\text{del}} \) in the flow table. Similarly to the \text{mod}
operation, \text{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 Open-
Flow 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 \( \prec \subseteq \text{Event} \times \text{Event} \)
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
\( \alpha \prec \pi \beta \) to denote that event \( \alpha \) occurs before event \( \beta \) in \( \pi \).
We use \text{HandleMsgs} to denote a set of all the events of
type \text{HandleMsg} and define such sets for each event
type defined in §3. We illustrate the HB ordering rules in-
duced from a given trace \( \pi \) in Fig. 2. All except four rules
(\text{BarrierPRE}, \text{BarrierPOST}, \text{Time1}, \text{Time2}) make use
of the information provided by the attributes \text{pid}, \text{out_pids},
\text{mid}, and \text{out_pids}. These capture the causality between
two events \( \alpha \) and \( \beta \) in the trace, where \( \alpha \) caused \( \beta \) to hap-
pen. \text{BarrierPRE}, \text{BarrierPOST} describe the effect of
\text{BARRIER_REQUEST} messages on OpenFlow switches. The
rules \text{Time1} and \text{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.

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

\text{SwitchBuffer}: When sending a \text{PACKET_IN} message to
the controller in a \text{SendMsg} event, the full packet con-
ten 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 \text{HandleMsg} event of
\text{msg_type} \text{PACKET_OUT} or \text{FLOW_MOD} may retrieve
the packet from the buffer before processing it. This rule orders
\text{HandlePkt} and \text{HandleMsg} events that store a packet in
the switch buffer before the \text{HandleMsg} event that eventually
retrieves a packet from the switch’s buffer. In the exam-
ple, this rule introduces the ordering \( 1 \prec 5 \).

\text{Host}: This rule orders the processing of the packet in a
\text{HostHandlePkt} event before the sending of the reply
packets in \text{HostSendPkt} events.

\text{Controller}: This rule orders the processing of the Open-
Flow message in a \text{CtrlHandleMsg} event before the sending
of the reply messages in \text{CtrlSendMsg} events. In the
example, this rule introduces the orderings $2 \prec 3$, $2 \prec 4$, $2 \prec 5$, $2 \prec 9$, and $2 \prec 10$.

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

**CONTROLPLANE TO** and **CONTROLPLANE FROM:** 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 $\delta$ 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 $\delta$ 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 $\delta$ 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 $\varphi$ over pairs of operations. For a pair of operations $a$ and $b$, the predicate $\varphi(a, b)$ evaluates to true if operations commute and to false otherwise.

**Auxiliary Relations** We define three auxiliary functions. First, we overload the set intersection operation $e_1 \cap e_2$ 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 $e_1 \subseteq e_2$ to model the semantics of table entry matching in regular and strict modes as follows:

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

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

$$
deletes(e, e, \text{strict}) :=
$$

$$
\begin{align*}
&\text{strict} \\
e &\subseteq e_{\text{del}} \land e.\text{out}_\text{port} \subseteq e_{\text{del}}.\text{out}_\text{port}
\end{align*}
$$

**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.

$\varphi(\text{add}, \text{add})$: 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.

$\varphi(\text{add}, \text{mod})$: 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.

$\varphi(\text{del}, \text{mod})$: 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.

$\varphi(\text{add}, \text{del})$: 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.

$\varphi(\text{mod}, \text{mod})$: 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.
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 check 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:

\[ R(\pi, \alpha) := \{ \beta \mid \beta \in \pi \land \alpha \prec \gamma_1 \prec \gamma_2 \prec \beta \land \gamma_1 \in CtrlHandleMsgs \land \gamma_2 \in CtrlSendMsgs \land \gamma_1, \gamma_2 \in Succ(\alpha) \land \beta \in Succ(\gamma_2) \} \]

where Succ(γ) 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 $\Gamma$ 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 $\tau$, we denote the set of commutativity races in $\tau$ as $CR(\tau)$.

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 $\Gamma^* = \{\Gamma_1, \Gamma_2, \Gamma_3, \ldots\}$ is isolated, by checking if no pair of events (across different policy changes) is in the set of commutativity races $CR(\pi)$:

$$UI(\Gamma^*) := \#\alpha, \beta : (\alpha, \beta) \in CR(\pi) \land \alpha \in \Gamma_u \land \beta \in \Gamma_v \land \Gamma_u \neq \Gamma_v$$

### 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 $\pi$, 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 $\pi$ that were created as a result of processing event $\gamma$. We say that event trace $\tau(\pi, \gamma)$ corresponds to a packet trace for a given packet $pkt$ if event $\gamma$ originated the packet $pkt$.

More formally, the event trace $\tau(\pi, \gamma)$ is defined as follows:

$$\tau(\pi, \gamma) := \gamma \cup \{\tau(\pi, \beta) \mid \beta \in Succ(\gamma) \land \beta \notin HostHandlePkts\}$$

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

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

We check packet coherence for all packets $pkt$ in a given trace $\pi$, i.e., we check coherence for each packet trace $\tau(\pi, \gamma)$ extracted from the trace $\pi$. We can be certain that a packet trace $\tau(\pi, \gamma)$ exhibits packet coherence if $CR_\tau(\pi, \gamma) = \emptyset$: 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 $CR_\tau(\pi, \gamma) \neq \emptyset$. 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 $sw$.

$$PC(\pi, \gamma) := CR_\tau(\pi, \gamma) = \emptyset$$

$$\lor(\exists e : (\forall(\alpha, \beta) \in CR_\tau(\pi, \gamma) : \alpha = e \land \forall(\alpha, \beta) \in CR_\tau(\pi, \gamma) : \beta.sw = e.sw))$$

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 $\pi$ may be free of violations, there may be another trace $\pi'$ with the same inputs as $\pi$ which does contain violations. In contrast, our checks on $\pi$ guarantee that any such trace $\pi'$ is free of violations, which is useful as it means we do not need to explore all possible traces $\pi'$. 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
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 moreHB 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 a 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).

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

<table>
<thead>
<tr>
<th>Controller</th>
<th>POX</th>
<th>FloodLight</th>
<th>ONOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC</td>
<td>40</td>
<td>139</td>
<td>55</td>
</tr>
</tbody>
</table>

**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.
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 filtering is the most efficient, followed by Time-based filtering.

Filter 2. Time-Based

Time filtering further helps reduce the number of races (Fig. 5) as the events relate to different hosts and non-overlapping entries.

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 (§5). 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 mod-
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.

![Figure 5: The effect of time-based filter. Choosing smaller $\delta$ filters more races. With $\delta=2$, SDNRacer can filter more than 20% of the races in 48% of the cases. While with $\delta=8$, SDNRacer can filter 20% of the races in 40% of the cases.](image)

For some 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 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.
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 checks 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


