Detecting Concurrency Violations in Software-Defined Networks

Author(s):
Miserez, Jérémie

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010437863

Rights / License:
In Copyright - Non-Commercial Use Permitted
Detecting Concurrency Violations in Software-Defined Networks

Jérémie Miserez

Master’s thesis
ETH Zurich, Department of Computer Science
April 2015

Supervisor:
Prof. Dr. Martin Vechev
Acknowledgements

I would like to thank both Prof. Dr. Martin Vechev and Prof. Dr. Laurent Vanbever for their expert knowledge, valuable ideas and discussions, and focus on seeing the bigger picture. Furthermore, I would like to express my sincere thanks to Pavol Bielik for his advice and valuable input on defining the details of the happens-before rules and the commutativity specification. Lastly, I would like to thank Ahmed El-Hassany for his technical expertise with the STS implementation.
Contents

Acknowledgments ii

1 Introduction 1

2 Motivating examples 2
  2.1 Stateful firewall allows traffic to be blocked 2
     2.1.1 Non-deterministic forwarding loop in a load balancer 3

3 The OpenFlow protocol 4
  3.1 OpenFlow switches 4
     3.1.1 Flow table 4
     3.1.2 Secure channel 6
     3.1.3 Sources of non-determinism inside a single switch 6
     3.1.4 OpenFlow messages 6
     3.1.5 OpenFlow controllers 7

4 Races in OpenFlow networks 8
  4.1 Defining the Happens-Before ordering 8
     4.1.1 Atomic operations and events 8
     4.1.2 Happens-Before rules 9
     4.1.3 Possible speculative rules 11
  4.2 Concurrency analysis 11
     4.2.1 Logical memory locations 11

5 Reducing the number of reported races 12
  5.1 Commutativity: Filtering harmless races 12
     5.1.1 Naive approach 12
     5.1.2 Fast approach 12
  5.2 Filtering harmless read-write races 15
     5.2.1 Read-write races without common ancestor 15
     5.2.2 Read-write races for packets of the same flow 15

6 Implementation 16
  6.1 Network and switch instrumentation 16
  6.2 Host instrumentation 16
  6.3 Controller instrumentation 16
  6.4 Race detector 17
     6.4.1 Constructing the Happens-Before graph 17
     6.4.2 Race detection 17
     6.4.3 Reducing the number of reported races 17
     6.4.4 Output 17
7 Evaluation
  7.1 Finding concurrency violations ........................................... 18
     7.1.1 Stateful firewall ............................................. 18
     7.1.2 Load balancer .................................................... 18
  7.2 Performance of the analyzer ......................................... 19

8 Related Work

9 Conclusion

Bibliography
Chapter 1

Introduction

Software-Defined Networking (SDN) holds a great promise for managing network complexity. The key idea of SDN is to enable logically centralized and direct control of the forwarding behavior of a network. However, realizing this vision relies on the difficult task of building highly sophisticated and reliable SDN control software. By design, SDN control software operates on top of a highly asynchronous environment: packets may arrive at a switch, links or nodes can fail, and expiring flows are be dispatched to the controller at any time.

Highly asynchronous programs are prone to concurrency errors caused by interfering access (a form of data races) to shared memory locations, and SDN control software is no exception. However, it is well known that discovering those errors is difficult as they often depend on a particular ordering of specific events. At the same time detecting these issues is important as they are often the root cause of underlying undesired behaviour.

This work presents techniques for automatically detecting concurrency errors arising in OpenFlow networks. Conceptually, there are two places where interference can occur: (i) within the control software itself (e.g. if it is multi-threaded), and (ii) between the control software and the network switches. Network switches can be seen as memory locations which are read and written to by various events. The first kind of interference can be detected with standard approaches, thus we focus on techniques specific to the second kind.

To capture asynchrony between a control program and the underlying network, we define a happens-before (HB) relation [9] for the most commonly used OpenFlow features. The HB relation succinctly captures the partial ordering of events in an OpenFlow network. In addition to the HB model, we present a commutativity specification of a network switch that precisely captures the conditions under which two operations on the switch commute. This specification elegantly abstracts the behaviors of the switch and is key to enabling precise analysis of the network.

Based on these models, we build a dynamic and controller-agnostic concurrency analyzer for SDN networks and use it to detect various data races in OpenFlow applications. The main contributions of this work are:

- A happens-before (HB) model capturing the asynchronous interaction between an OpenFlow-based SDN controller and the underlying devices.

- A commutativity specification capturing the precise conditions under which two high-level operations on an OpenFlow switch commute.

- An implementation of a dynamic analyzer which uses the HB model and the commutativity specification to automatically detect concurrency violations occurring on network switches.

- An evaluation indicating that the tool is able to uncover concurrency issues leading to harmful behaviors.
Chapter 2

Motivating examples

In this section, we explain in more detail how concurrency errors can arise in SDN with two motivating examples. This is intended to give a short overview of the topic, a more detailed description of the OpenFlow protocol follows in Section 3.

A SDN controller is an event-driven program in which events can occur both asynchronously (a packet received from the network, a link failure) or synchronously (as a result of a request issued by the controller). A SDN controller basically writes to and reads from the flow table of switches.

The flow table of a switch is an ordered list of forwarding entries against which packets are being matched and the corresponding forwarding action is taken. As such, a SDN switch can be thought of as a separate application that reads from the flow table whose state is written and queried by the controller.

In the following, we consider that a concurrency violation arises whenever we encounter two unordered accesses to the switch flow table, one of which must be a write produced by the controller.

2.1 Stateful firewall allows traffic to be blocked

Consider a controller program running a stateful firewall, as shown in Fig. 2.1, that allows internal hosts to initiate communication with external hosts, but blacklists external hosts from sending unsolicited traffic to internal hosts.

A possible sequence of events is shown in Fig. 2.1. Host 1 sends a packet 1 to Host 2 which hits the switch 2 and is sent to the controller 3. Since the communication is initiated by internal host, the controller pushes down two FLOW_MOD rules and sends a PACKET_OUT message instructing the switch
to forward the packet. Because the switch is allowed to execute messages out of order, it handles the PACKET_OUT message $\texttt{4}$ first and sends $\texttt{5}$ the packet further to Host 2. The Host 2 receives the packet $\texttt{6}$ and responds immediately with a packet $\texttt{7}$ that hits the switch back $\texttt{8}$ before the rules have been installed. Consequently, the return packet goes to the controller $\texttt{9}$. In the meantime, the two rules enabling the bi-directional communication are installed $\texttt{10}$–$\texttt{11}$. As the return packet comes from Host 2, the controller instructs the switch to install a drop rule $\texttt{12}$ which drops the communication.

In this example, there exists a concurrency error due to non-deterministic order between the write event $\texttt{13}$, the installation of the rule matching packets from Host 2, and the read event $\texttt{6}$, the reception by the switch of the return packet from Host 2. A simple fix is for the controller to issue a BARRIER message after the two rules installation request and before sending the packet out to Host 2.

### 2.2 Non-deterministic forwarding loop in a load balancer

![Diagram](image)

Figure 2.2: An example of a simple load-balancing application (right) and a sequence of events (left), which leads to a forwarding loop.

In this example we consider a controller that is running a simple load-balancer application, Fig. 2.2. Consider the sequence of events shown in Fig. 2.2: 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 switch $\texttt{1}$ and is sent to the controller $\texttt{2}$. The controller elects Replica#1, computes the shortest-path between $\text{S}1$ and $\text{S}2$, pushes down two FLOW_MOD on $\text{S}1$ and $\text{S}2$, and sends a PACKET_OUT message instructing $\text{S}1$ to forward the packet to $\text{S}2$. $\text{S}1$ installs the rule $\texttt{3}$ and handles the message PACKET_OUT $\texttt{4}$ that it sends to $\text{S}2$ $\texttt{5}$. The packet hits $\text{S}2$ $\texttt{6}$ before the corresponding flow rule is installed $\texttt{8}$ and is sent back to the controller $\texttt{7}$. Assuming a round-robin selection algorithm, the controller now elects Replica#2, computes the shortest-path between $\text{S}2$ and $\text{S}3$ and pushes down the corresponding flow rules on $\text{S}2$, $\text{S}1$ and $\text{S}3$. From this point on, the traffic is being processed in a non-deterministic manner as $\text{S}1$ and $\text{S}2$ each have two rules with the same priority that match each direction of the traffic. Concretely, the traffic either ends up in a forwarding loop, if $\text{S}1$ uses the rule to forward the traffic to $\text{S}2$, and vice-versa or hits one of the two replica (again non-deterministically).

In this example, the concurrency error arises between the read event $\texttt{6}$, the outbound packet received by $\text{S}2$ and the write event $\texttt{8}$ on $\text{S}2$ matching it.
Chapter 3

The OpenFlow protocol

In this section we provide a brief overview of the parts of an OpenFlow network that are directly relevant to our goal of finding concurrency violations.

The OpenFlow protocol decouples the control and data planes: Compared to conventional switches, OpenFlow switches are reduced to only do packet forwarding and packet modification according to a set of rules stored locally per switch. If an incoming packet does not match any of the existing rules it is sent to the controller for further processing. The controller has a centralized view of the network and may then modify and forward the packet, as well as install or modify rules in any of the switches in the network. Ideally, the controller will install rules so that subsequent similar packets can be processed directly by the switch without having to go through the controller. The main value of OpenFlow lies in the ability to run any number of custom software applications on the controller, and in the ability of the controller to see a complete, centralized view of the network.

The term SDN is also frequently used to refer projects other than OpenFlow, however in this work we will focus solely on OpenFlow itself. Precise definitions of SDN terminology, an account of the history of SDN, as well as an overview of current SDN projects can be found in [1], a comprehensive survey on the topic.

The OpenFlow Switch specification v1.0.0 [2] and accompanying errata v1.0.2 [3] describe the requirements switches need to fulfill to be considered OpenFlow compatible, as well as the communications interface and wire protocol used to communicate between the control plane (controller) and the data plane (switches). Unless otherwise noted the OpenFlow version used in this work is always v1.0.2, although later versions of the specification exist: at the time of writing the latest is v1.5.0 [4]. Newer versions are only slowly being supported by switch vendors and software frameworks and although the basic principles have remained the same, considerable additional complexity is being added with each new version.

3.1 OpenFlow switches

In this section we will describe the basic operation of an OpenFlow switch and introduce the notation used throughout this document.

3.1.1 Flow table

A basic component of each OpenFlow switch is the flow table. This table is responsible for performing packet lookups and packet forwarding. The flow table contains a set of entries (“flow”) used to match incoming packets.
Packet
Each packet contains a header and the 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 \( pkt \).

Flow table entry
Each flow table entry contains the fields match, priority, actions, as well as several other fields summarized here as counters. For flow table entry \( e \) we use notation \( e.m, e.p \) and \( e.a \) to refer to the match, priority and actions respectively.

Priority is a number specifying entry preference in case the packet matches multiple flow entries. Match can be either exact match or wildcard match. 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 packet and flow table entry. Wildcards allow a single entry to match several concrete packet headers: e.g. if the match field \( e_1.m \) contains the wildcard \( 10.0.*.* \) for the IP source field, then \( e_1.m \) will match all values ranging from \( 10.0.0.0 \) to \( 10.0.255.255 \).

Actions specify a set of forwarding and modification operations to be performed on a matching packet. Counters are used for various purposes. There are statistics that track how many packets and bytes have been matched to each rule, as well as timer statistics that allow removing a rule after a certain fixed amount of time or after a certain period of inactivity.

<table>
<thead>
<tr>
<th>#</th>
<th>Priority</th>
<th>Match</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>IP src</td>
<td>IP dst</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10.0.0.2</td>
<td>10.0.0.1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>10.0.<em>.</em></td>
<td>10.0.<em>.</em></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>10.0.0.*</td>
<td>10.0.0.*</td>
</tr>
</tbody>
</table>

Table 3.1: Sample flow table with three flow table entries.

Flow table operations
There are four types of operations that can be performed on the flow table. A read operation is performed on each received packet while add, mod and del are issued by the controller using a FLOW_MOD message.

read(pkt)/\( e_{\text{read}} \)
The read operation denotes matching of packet \( pkt \) against the flow table to determine the highest priority flow table entry \( e_{\text{read}} \) that should be applied.

add(\( e_{\text{add}}, \text{no\_overlap} \))
An add operation tries to add a new entry \( e_{\text{add}} \) 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.

mod(\( e_{\text{mod}}, \text{strict} \))
A mod operation modifies existing entries in the flow table. A boolean flag strict is used to distinguish between two types of modifications issued by the controller: MODIFY and MODIFY STRICT. In strict mode, 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 is also allowed to add entry in case no match is found.

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

The secure channel is a persistent TCP connection between the switch and the controller, over which OpenFlow messages are sent to and from the switch. For OpenFlow v1.0.2, there is exactly one connection per switch as multiple controllers are not supported. The connection for the secure channel is always out-of-band, and messages on this channel arrive in the order in which they are sent. All messages sent on the secure channel are in the OpenFlow wire format.

3.1.3 Sources of non-determinism inside a single switch

There are several sources of non-determinism inside the switch:

Packet processing

Packets received by the switch may be processed in parallel. This allows improved performance for hardware switches as well as software switches when run on multicore processors.

OpenFlow message reordering

OpenFlow messages received by the switch may be executed in any order as long as no explicit synchronization message (BARRIER_REQUEST message) is received. This means that although messages arrive at the switch in the same order as they were sent from the controller, they may not necessarily be executed in that order.

Output QoS queuing

Each OpenFlow switch has several user-definable QoS packet output queues. It is possible to set a desired send rate for each queue, in OpenFlow v1.0.2 it is however not possible to ensure that a given packet/queue is sent before any other queue. In practice this means that packets may be reordered before being sent.

3.1.4 OpenFlow messages

Only a subset of OpenFlow messages are relevant to finding concurrency violations.

3.1.4.1 Switch-to-controller messages

These messages can be sent by the switch any time when the controller needs to react to events on the switch.

Packet-in (PACKET_IN)

When a packet received by the switch does not match any of the existing rules, it is buffered at the switch and a PACKET_IN message consisting of the packet header and auxiliary information such as the buffer ID is sent to the controller. If all buffers on the switch are full the full packet is included.

Flow-Removed (FLOW_REMOVED)

Flows with a timeout value may be configured to trigger a FLOW_REMOVED message when they expire. Such a message may also be generated when a flow is deleted by the controller.

Port-Status (PORT_STATUS)

A PORT_STATUS message is generated when one of the ports on the switch is modified, e.g. by plugging in or unplugging a cable.
3.1.4.2 Controller-to-switch messages

This type of message can be sent by the controller at any time, for any reason. Some of these will generate a response message from the switch after execution.

Modify-State (FLOW_MOD, PORT_MOD)

FLOW_MOD messages allow adding, deleting, or modifying the flow table of a switch. PORT_MOD messages allow enabling/disabling physical ports. Interestingly, a FLOW_MOD that adds or modifies flows may also include an optional parameter for a packet buffer id. If such a parameter is given, the switch will process the flow table modification and then immediately afterwards process the packet in the given buffer. Note that this parameter is unsupported when flows are deleted by the FLOW_MOD message, as this might result in FLOW_REMOVED messages.

Send-Packet (PACKET_OUT)

Sending a PACKET_OUT message allows sending a given packet out a specified port on the switch.

Barrier (BARRIER_REQUEST)

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, before executing any messages beyond the BARRIER_REQUEST. The switch will send a BARRIER_RESPONSE after all pending messages and the BARRIER_REQUEST message have been processed.

3.1.5 OpenFlow controllers

A multitude of OpenFlow controllers are available, running various different applications. In this work we are using the POX [5] and Floodlight [6] controllers. There are two types of controller applications: Reactive applications are event-driven and react to messages sent from the switch to the controller. Whenever such a message is received, they may take one or several actions. Proactive applications may take actions based on external events, such as timers or commands from a network administrator. Most software frameworks allow composition of multiple small applications, thus in practice controllers usually run both reactive and proactive components.

Sources of non-determinism inside controllers

As OpenFlow controllers may run applications written in any programming language and may run on any hardware they have the same sources of non-determinism as regular applications. This means that a multithreaded controller running on a multiprocessor system might inadvertently reorder packets or execute event handlers concurrently. In OpenFlow v1.0.2 this problem is slightly reduced due to the fact that only a single controller is supported, whereas later versions of OpenFlow bring support for more than one controller.

POX and Floodlight

Applications built on top of the POX or Floodlight frameworks have stronger guarantees. As POX is a single-threaded controller, applications running on POX will not exhibit errors such as the ones described above. Floodlight never uses more than one thread per switch, which means that incoming messages from a single switch will never be executed concurrently or out of order. Note however that this means that applications written for Floodlight will need to synchronize accesses to external databases or shared datastructures as soon as more than one switch is used. Finding such race conditions inside the controller applications can be accomplished by using any number of analysis tools and is out of scope for this work.
Chapter 4

Races in OpenFlow networks

We present a formal model capturing the asynchrony arising in OpenFlow. Developing a concurrency analyzer requires the following building blocks:

1. A definition of the (potentially concurrent) operations and atomic events.
2. A definition of the happens-before (HB) relation between the events.
3. A definition of when two concurrent events interfere.

4.1 Defining the Happens-Before ordering

Our goal was to model the ordering of the events as precisely as possible, while at the same time having a succinct and precise model. For example, one could simply define the events at finer granularity (such as write packet to a buffer or a read packet from buffer). However, such a definition would be more difficult and cumbersome to work with as it would expose internal implementation details of the switch (that might differ between implementations) and contain more events and happens-before rules. The challenge therefore is in designing an abstraction that captures all the relevant information for the concurrency analysis and can be described using small set of events and happens-before rules.

Our HB relation is based on in-depth studying of the OpenFlow specification as well as the behaviour of actual network switches [7] and associated design documents [8] where necessary. However, it should be noted that all events, operations and HB rules as presented here are not specific to a certain controller or switch but are general enough to cover all OpenFlow compliant switches and controllers.

4.1.1 Atomic operations and events

We begin by defining a small set of events, denoted as $\textit{Event}$, which succinctly encapsulate the relevant operations performed by network switches and hosts in the network. The operations are exactly the ones defined in §3.1.1 and contain one or multiple reads and writes to the flow table. For each event, we define a set of attributes that describe the event and are later used to build the HB ordering. The set of attributes is as follows:

\[
\langle \text{pid\_in}, \text{pid\_out}, \text{mid\_in}, \text{mid\_out}, \text{msg\_type}, \text{switch\_id} \rangle
\]

where, $\text{pid\_in}$ and $\text{pid\_out}$ denote an identifier assigned to a packet, $\text{mid\_in}$ denotes a single identifier assigned to an OpenFlow message, $\text{mid\_out}$ denotes a set of identifiers corresponding to a set of OpenFlow messages and $\text{switch\_id}$ is a switch identifier. The $\text{msg\_type}$ denotes an OpenFlow message type where the relevant types for concurrency analysis are $\text{PACKET\_IN}$, $\text{PACKET\_OUT}$, $\text{BARRIER\_REQUEST}$, $\text{BARRIER\_REPLY}$, $\text{PORT\_MOD}$, $\text{FLOW\_MOD}$, and $\text{FLOW\_REMOVED}$. The $\text{out}$ keyword for identifiers denotes that the instrumentation always generates a new identifier whereas the $\text{in}$ keyword
denotes that the previously generated identifier is used. We discuss the challenges in designing mechanisms for generating unique identifiers for packets in Section 6.1. Depending on the event type, only a subset of attributes is used. The behavior of switches with switch_id, hosts and controllers is captured with the following events:

**PacketHandle** \((\text{pid}_\text{in}, \text{pid}_\text{out}, \text{mid}_\text{out})\)

- \(\text{PacketHandle}(\text{pid}_\text{in}, \text{pid}_\text{out}, \text{mid}_\text{out})\) denotes that the switch processed a data plane packet \(\text{pid}_\text{in}\). As a result of processing this packet, either an OpenFlow message is generated and sent to the controller (in which case \(\text{mid}_\text{out}\) contains a message identifier and \(\text{pid}_\text{out}\) is the identifier of the packet stored in the buffer) or the packet is forwarded (in which case \(\text{pid}_\text{out}\) contains the new packet identifier).

**PacketSend** \((\text{pid}_\text{in}, \text{pid}_\text{out})\)

- \(\text{PacketSend}(\text{pid}_\text{in}, \text{pid}_\text{out})\) denotes that the switch takes the packet \(\text{pid}_\text{in}\) and sends it out as \(\text{pid}_\text{out}\).

**MsgSend** \((\text{mid}_\text{in})\)

- \(\text{MsgSend}(\text{mid}_\text{in})\) denotes that an OpenFlow message with \(\text{mid}_\text{in}\) was sent to the controller.

**MsgHandle** \((\text{mid}_\text{in}, \text{pid}_\text{in}, \text{pid}_\text{out}, \text{mid}_\text{out}, \text{msg}_\text{type})\)

- \(\text{MsgHandle}(\text{mid}_\text{in}, \text{pid}_\text{in}, \text{pid}_\text{out}, \text{mid}_\text{out}, \text{msg}_\text{type})\) denotes that the OpenFlow message \(\text{mid}_\text{in}\) whose type is \(\text{msg}_\text{type}\) from the controller was processed. By default, the attributes \(\text{pid}_\text{in}, \text{pid}_\text{out}, \text{mid}_\text{out}\) are initialized to an undefined value \(\bot\) which denotes that the message did not generate any further events (e.g., it simply updated the flow table and finished). However, depending on how the message was processed it might generate other events, in which case the attributes are updated with values other than \(\bot\).

**HostHandle** \((\text{pid}_\text{in}, \text{pid}_\text{out})\), **HostSend** \((\text{pid}_\text{in}, \text{pid}_\text{out})\)

- \(\text{HostHandle}(\text{pid}_\text{in}, \text{pid}_\text{out})\), \(\text{HostSend}(\text{pid}_\text{in}, \text{pid}_\text{out})\)
  denote packet receive and send by a host respectively. Note that assigning correct values to the \(\text{pid}_\text{out}\) value of HostHandle and the \(\text{pid}_\text{in}\) value of HostSend requires instrumentation of the host.

**ControllerHandle** \((\text{mid}_\text{in}, \text{mid}_\text{out})\), **ControllerSend** \((\text{mid}_\text{in}, \text{mid}_\text{out})\)

- \(\text{ControllerHandle}(\text{mid}_\text{in}, \text{mid}_\text{out})\), \(\text{ControllerSend}(\text{mid}_\text{in}, \text{mid}_\text{out})\)
  denote message receive and send by a controller respectively. Note that assigning correct values to the \(\text{mid}_\text{out}\) value of ControllerHandle and the \(\text{mid}_\text{in}\) value of ControllerSend requires instrumentation of the controller.

### 4.1.2 Happens-Before rules

The happens-before [9] relation \(<\subseteq Event \times Event\) is a binary relation that is irreflexive and transitive. For convenience, we use \(\alpha < \beta\) instead of \((\alpha, \beta) \in<\). For each finite trace \(\pi = \alpha_0 \cdot \alpha_1 \cdot \cdots \cdot \alpha_n\) consisting of a sequence of events we use \(\alpha <_\pi \beta\) to denote that event \(\alpha\) occurs before event \(\beta\) in \(\pi\). This order records causal relationships between events, i.e. if there is a HB relationship between two events \(\alpha < \beta\), then \(\alpha <_\pi \beta\) always holds. If there is no relationship (neither \(\alpha < \beta\) nor \(\beta < \alpha\)), then they may be processed concurrently, denoted by \(\alpha \parallel \beta\). We formalize the HB rules in Fig. 4.1, an description of each rule follows.

#### Packets

This rule orders events that handle the packet as it moves through the network, i.e. a packet is sent before it is received. Each event handling a packet \(\text{pid}_\text{in}\) generates new, unique \(\text{pid}_\text{out}\). This guarantees that there is at most one pair in the trace for which \(\text{pid}_\text{in} = \text{pid}_\text{out}\). There could be no pair in case the packet is dropped. In Fig. 2.1, this rule introduces the orderings \(1 < 2, 4 < 5, 5 < 6\) and \(7 < 8\).

#### Messages

This rule orders messages sent to the controller from the switch with the responses the controller sends back and the other way around. Note that there could be multiple MsgHandle with the same \(\text{mid}_\text{in}\) in case the controller sends multiple messages in the response. The mapping from \(\text{mid}_\text{in}\) to \(\text{mid}_\text{out}\) must be captured separately by the controller instrumentation. In Fig. 2.1, this rule introduces the orderings \(2 < 3, 3 < 4, 3 < 10, 3 < 11, 6 < 9\) and \(9 < 12\).
Packets:
\[ \alpha \in \{\text{PacketHandle}, \text{PacketSend}, \text{MsgHandle}, \text{HostSend}\} \]
\[ \beta \in \{\text{PacketHandle}, \text{PacketSend}, \text{HostHandle}\} \]
\[ \alpha.\text{pid}_{\text{out}} \neq \perp \quad \beta.\text{pid}_{\text{in}} \neq \perp \quad \alpha.\text{pid}_{\text{out}} = \beta.\text{pid}_{\text{in}} \]
\[ \alpha \prec \beta \]

Messages:
\[ \alpha \in \{\text{PacketHandle}, \text{MsgHandle}, \text{MsgSend}\} \]
\[ \beta \in \{\text{MsgHandle}, \text{MsgSend}\} \]
\[ \alpha.\text{mid}_{\text{out}} \neq \perp \quad \beta.\text{mid}_{\text{in}} \in \alpha.\text{mid}_{\text{out}} \]
\[ \alpha \prec \beta \]

FlowRemoved:
\[ \alpha = \text{MsgHandle} \quad \alpha.\text{msg}_{\text{type}} = \text{FLOW_MOD} \]
\[ \beta = \text{MsgSend} \quad \beta.\text{msg}_{\text{type}} = \text{FLOW_REMOVED} \quad \alpha <_\pi \beta \]
\[ \alpha.\text{switch}_{\text{id}} = \beta.\text{switch}_{\text{id}} \quad \alpha.\text{match} = \beta.\text{match} \]
\[ \alpha.\text{priority} = \beta.\text{priority} \quad \alpha.\text{cookie} = \beta.\text{cookie} \]
\[ \alpha \prec \beta \]

BarrierPre:
\[ \alpha, \beta \in \text{MsgHandle} \]
\[ \alpha.\text{switch}_{\text{id}} = \beta.\text{switch}_{\text{id}} \quad \beta.\text{msg}_{\text{type}} = \text{BARRIER_REQUEST} \quad \alpha <_\pi \beta \]
\[ \alpha \prec \beta \]

BarrierPost:
\[ \alpha, \beta \in \text{MsgHandle} \]
\[ \alpha.\text{switch}_{\text{id}} = \beta.\text{switch}_{\text{id}} \quad \alpha.\text{msg}_{\text{type}} = \text{BARRIER_REQUEST} \quad \alpha <_\pi \beta \]
\[ \alpha \prec \beta \]

Host:
\[ \alpha \in \text{HostHandle} \quad \beta \in \text{HostSend} \quad \alpha.\text{pid}_{\text{out}} = \beta.\text{pid}_{\text{in}} \]
\[ \alpha \prec \beta \]

Figure 4.1: Happens-before rules capturing ordering of packets an OpenFlow messages in SDN network.

FlowRemoved
This rule captures the fact that the switch can send a FLOW_REMOVED message only after the corresponding flow was added to the table using the FLOW_MOD message, i.e. a flow cannot be removed before it was added.

BarrierPre
This rule orders messages received before a BARRIER_REQUEST with the BARRIER_REQUEST. All such messages must be processed before the BARRIER_REQUEST is processed.

BarrierPost
This rule orders messages received after a BARRIER_REQUEST with the BARRIER_REQUEST. All such messages must be processed after the BARRIER_REQUEST is processed.

Host
There is an ordering between the host receiving a packet and responding to it. Note that this must be captured by the Host instrumentation. In Fig. 2.1, this rule introduces the ordering \(6 < 7\).
4.1.3 Possible speculative rules

The rules so far make no assumptions that could be leveraged to add additional rules. It is however possible to think of such rules for certain situations. Note however that we do not use any such rules in this work.

Hosts
Instead of instrumenting hosts, it could be assumed that all packets sent and received by the host are in trace order, i.e. each incoming packet is ordered before all successive outgoing packets by the same host. This is true for certain applications (e.g. single-threaded web servers), but not in general.

Switch hardware
In addition to instrumenting switches it could be assumed that messages/packets are never reordered inside the switch, or only under certain circumstances. E.g. it might be known that a switch never reorders messages more than n messages apart, or that a message is always processed in a certain window of time when the switch speed and queue lengths are known.

4.2 Concurrency analysis

We define two events to interfere if they can occur concurrently, both access the same memory location, and at least one of the accesses is a write. In our case memory locations are the flow tables of each switch.

4.2.1 Logical memory locations

Switch flow tables
Regular addressable shared memory as it is found on many systems consists of many memory locations. Each of these locations can be accessed independently from one another. However, in the case of OpenFlow switch flow tables this is not possible. Even though each flow table internally consists of a separate memory location for each flow, the nature of the operations supported by OpenFlow switches means that the flow table as a whole should be viewed as a single data structure. Therefore the only type of memory location that we can use for the purposes of race detection is the unique switch id.

Switch packet buffers
A secondary type of memory location are the packet buffers in each switch that are used for buffering packets when a PACKET_IN message is sent to the controller. These buffers are then read when a PACKET_OUT or a FLOW_MOD message is processed and the packet is retrieved from the buffer and forwarded. Due to how these buffers are implemented in the switch, the controller only has control over the reads to each buffer and not the writes: writes will automatically only occur to buffers that are free, and this is enforced by the switch hardware. Thus, these buffers are not of interest for our race detection.
Chapter 5

Reducing the number of reported races

Running a race detector that just makes use of the HB rules leads to a lot of races. To address this issue we define three techniques to filter harmless races which are described next. Write-Write conflicts occur whenever two FLOW_MOD messages modify the same flow table and there is no BARRIER_REQUEST separating them. Read-Write conflicts occur when an incoming packet or a PACKET_OUT message is processed by the switch and there is no ordering between the packet and all FLOW_MOD messages previously processed on the switch.

5.1 Commutativity: Filtering harmless races

It is clear that viewing each switch’s flow table as just one single memory location results in many reported races, even when reordering the operations involved in the race would not actually change the outcome of either operation. Intuitively, commutativity captures whether changing the order of two operations affects the computation result. The computation results include relevant flow table state together with the return values (if any) of the participating operations. We consider two flow tables to be in the same state if they contain the same set of flow table entries, except for counters which are ignored. Races between commuting operations are harmless, while races between non-commuting operations are potentially harmful and indicators of bugs.

5.1.1 Naive approach

It is relatively easy to determine if two operations commute: For each operation, we store the state of the flow table before the operation and the state of the flow table after the operation. Then, in order to determine if two operations commute both orderings are simulated and the respective outputs and flow tables compared. Although both sound and precise, this method has considerable overhead both in computation time and storage space.

5.1.2 Fast approach

A better option is to determine commutativity directly from the operations themselves. The commutativity specification is conveniently specified in the form of a predicate $\varphi$ over pairs of operations using formulas written in propositional logic. For a pair of operations $a$ and $b$, the predicate $\varphi_{ab}$ evaluates to true if operations commute and to false otherwise.
CHAPTER 5. REDUCING THE NUMBER OF REPORTED RACES

13

The 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. This allows reducing the

\[ \varphi_{\text{read}}(\text{pkt})/\text{read} \]  
\[ \varphi_{\text{add}}(\text{e, na_overlap}) := \neg(e_{\text{read}} = e_{\text{add}}) \]  
\[ \neg(pkt.h \subseteq e_{\text{add}}.m \land e_{\text{read}}.p \leq e_{\text{add}}.p \land e_{\text{read}}.a \neq e_{\text{add}}.a) \]  
\[ \text{if add} < \text{read} \]  
\[ \text{if read} < \text{add} \]  

\[ \varphi_{\text{mod}}(e_{\text{mod}}, \text{strict}) := \neg(e_{\text{read}} \subseteq e_{\text{mod}} \land a_{\text{read}}.a = e_{\text{mod}}.a) \]  
\[ \neg(pkt.h \subseteq e_{\text{mod}}.m \land e_{\text{read}}.a \neq e_{\text{mod}}.a) \]  
\[ \text{if mod} < \text{read} \]  
\[ \text{if read} < \text{mod} \]  

\[ \varphi_{\text{mod}}(\text{e_del, strict}) := \neg((\text{deletes}(e_{\text{del}}, e_{\text{read}}, \text{strict})) \land \text{strict_mod}) \]  
\[ \text{otherwise} \]  

\[ \varphi_{\text{add}}(e, \text{na_overlap}) := \neg((\text{deletes}(e_{\text{del}}, e, \text{strict}) \lor (\text{na_overlap} \land e \cap e_{\text{del}} \neq \emptyset)) \]  

\[ \varphi_{\text{del}}(e_{\text{del}}, \text{strict}) := \neg(e_{\text{add}} \subseteq e_{\text{mod}} \land a_{\text{add}}.a \neq e_{\text{mod}}.a) \]  
\[ \neg(e_{\text{add}} \cap e_{\text{mod}} \neq \emptyset) \]  
\[ \text{if } \neg\text{no_overlap} \]  
\[ \text{otherwise} \]  

\[ \varphi_{\text{add}}(e_{\text{e_del, no_overlap}}) := \neg(e_{\text{mod}}.m \cap e_{\text{mod}}.m \neq \emptyset) \]  

Figure 5.1: Commutativity specification of an OpenFlow switch. Note that two read and two del operations always commute.

5.1.2.1 Auxiliary functions

To avoid clutter we define three auxiliary functions. First, we overload the set intersection operator \( e_1 \cap e_2 \) for two entry match structures (or packet headers) and use it to compute all packet headers that may match both. This can easily be done even when wildcards are present [10]. Next, we use \( e_1 \subseteq e_2 \) to model the semantics of the table entry matching in the strict mode, defined as follows:

\[ e_1 \subseteq e_2 := \begin{cases} e_1.m = e_2.m \land e_1.p = e_2.p \quad \text{if } \text{strict} \\ e_1.m \subseteq e_2.m \quad \text{if } \neg\text{strict} \end{cases} \]

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

\[ \text{deletes}(e_{\text{del}}, e, \text{strict}) := \begin{cases} e \subseteq e_{\text{del}} \land e.\text{out_port} \subseteq e_{\text{del}}.\text{out_port} \end{cases} \]

5.1.2.2 Commutativity specification

The commutativity specification of an OpenFlow switch is shown in Fig. 5.1.
number of reported conflicts by not including operations that commute in the current flow table state but might not necessarily commute in all possible states.

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 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 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, the 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 the mod can add a new entry that overlaps with add which would result in add not being added.

\[ \varphi(\text{del}, \text{mod}) \]

If modify can affect 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.

\[ \varphi(\text{add}, \text{del}) \]

The 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.

\[ \varphi(\text{read}, \text{add/mod/del}) \]

For read operations we distinguish two cases depending on the order in which the operations are executed in the trace. If the read happens first, the operations do not commute if the matched entry is not guaranteed to match after second operation is performed. Since we know the concrete flow entry that matched the initial read, such 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 the delete operation, we conservatively check whether an entry that matches the packet can be removed.

5.1.2.3 Limitations

The read operation as defined in 3.1.1 returns the matched flow as a return value. This exposes some of the internal state of the flow table and allows verifying whether the read operation commutes. All other operations do not return any hidden state which significantly reduces the complexity of the instrumentation. However, this also leads to the issue of impreciseness: The specification might say that two operations do not commute when in fact they do [11]. Note that the opposite is not possible, i.e. if the predicate evaluates to true then the two operations must commute (soundness). An example for impreciseness can be easily constructed.
Example where the commutativity specification is imprecise

<table>
<thead>
<tr>
<th></th>
<th>Priority</th>
<th>Match</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>10.0.0.1</td>
<td>Fwd: port 1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10.0.0.1</td>
<td>Fwd: port 1</td>
</tr>
</tbody>
</table>

Table 5.1: Initial flow table with two rules

Consider the case where the initial flow table looks like Table 5.1. Then, two operations are executed: i) a `read(10.0.0.1)` and ii) a `del(10.0.0.1)`.

In both cases, the action taken is the same: the packet is forwarded out port 1. However, the commutativity specification fails to capture this. Without exposing significant amounts of hidden state, there is no way of knowing that the next-best match for the `read` operation after the deletion would result in the same action being taken. Thus the commutativity specification will return false regardless.

Note that in the worst case the amount of hidden state that would need to be returned for the `delete` operation would consist of the whole flow table. Knowing this, we can use the specification to efficiently check for operations that commute, however for the verification of non-commutativity the naive approach is still needed.

Also of interest is that the e.g. two identical `add` operations will not commute. In practice the flow table will end up containing the same entries, except for possible statistics counters and OpenFlow cookie values. Although this is often done in practice, the operations still interfere with each other and thus we feel it would not be correct to say that they commute. The abstract state of a flow table is simply the set of all rules contained in the table.

5.2 Filtering harmless read-write races

While commutativity checking reduces the number of reported potentially harmful races significantly, it still leaves users with a large number of races. Thus it is helpful to filter out certain specialized types of races, if we know they are most likely not indicators of bugs.

5.2.1 Read-write races without common ancestor

Large numbers of potentially harmful read-write races are reported when new packets are injected into the network. The read operations that follow when one of these packets enters a switch are usually not ordered with the write operations previously applied to the switch. This results in races each time the packet is processed by a switch. Although these races are valid, they are mostly useless for debugging.

We filter these races by hiding all races between operations that have no common ancestor in the HB graph. This allows debugging of interactions with packets and reactive controllers, but not of interactions with stray packets in the network. However, enabling this filter will remove races that might be relevant to developers debugging proactive controllers.

5.2.2 Read-write races for packets of the same flow

For reactive controllers, a special case of read-write race is that subsequent packets of the same flow race with the previously sent packets. Thus it is possible that the flow table read operation of a newer packet races with the installation of a flow table entry triggered by the older packet. This is a special case of a read-write race without a common ancestor and is handled the same way.
Chapter 6

Implementation

The implementation of our system consists of several parts: i) an instrumentation of the SDN troubleshooting system STS [12], ii) an partial instrumentation of the POX and Floodlight frameworks, and iii) a concurrency analyzer that implements the happens-before rules, commutativity checks, and filtering.

We make use of STS’s internal publish/subscribe event system to generate custom events for all individual actions and events taking place within the network. Multiple of these small custom events are then combined to generate larger events as defined in Section 4.1.1. These larger events are then converted to a common format and the operations associated with each event are serialized to a JSON format and written to a file. This decouples the race detection from the simulation and allows us to run the detector both offline and online. The serialization step aids in debugging and prevents side effects due to accidental modification of STS data structures by the race detector.

6.1 Network and switch instrumentation

We use STS to simulate a complete network, including OpenFlow switches, wires, and hosts. This allows tracking the path each packet takes through the switches and hosts. Tracking packets through a network without ambiguity presents a challenge [13], as approaches that rely on using the packet itself fail in the presence of duplicate identical packets (from the same host or from different hosts) or when packets are modified inside switches. Tagging or encapsulating packets has an effect on how the packet is handled in the network, and is not applicable in all situations. In our case, it was not acceptable to have any ambiguity in tracking packets through the network. However, as STS is a network simulator it is possible to accurately track packets without ambiguity by using the Python object ids and a separate bookkeeping data structure. This enables us to track packets even when they are modified within switches. We instrument the switches within STS to store snapshots of flow tables before and after each read or write operation.

6.2 Host instrumentation

STS features simulated hosts that respond to ARP queries. These hosts were modified to respond to ICMP pings and instrumented to capture the order between HostSend and HostHandle events.

6.3 Controller instrumentation

Both POX and Floodlight controllers are partially instrumented: The instrumentation includes a wrapper around the event handler for incoming messages, and adding happens-before relations automatically whenever a message is sent, thus capturing the order between MsgSend and MsgHandle events. We note that this approach does not capture happens-before relations introduced by explicit synchronization used inside the controller. It is however sufficient for reactive controllers, such as the
ones used by our examples, since messages sent out by the controller are always a direct result of handling an incoming OpenFlow message.

Due to the fact that the controller instrumentation is not part of STS, the controller instrumentation writes the incoming and outgoing OpenFlow messages to a log file in Base64, along with the happens-before relation between each pair. The STS instrumentation periodically reads this log and matches up the incoming and outgoing messages with the ones from the controller instrumentation. Internally, two new events are created to make edges added by the controller instrumentation explicit: ControllerHandle and ControllerSend. When matching up Base64 values with OpenFlow messages, care needs to be taken to not inadvertently modify or deserialize the messages after receiving them, as even the slightest change in a message will make it impossible to match it with its Base64 representation gathered from the controller.

The instrumentation for Floodlight consists of a module. When the event handler is invoked for a message of type PACKET_IN, FLOW_REMOVED, BARRIER_REPLY, or PORT_MOD, it will store the incoming message in a variable only valid in the context of the current event handler. When an outgoing message is sent and this variable contains an incoming message, the happens-before relation is added.

The POX instrumentation uses the same basic method, but instead of an added module we modified a version of POX directly.

### 6.4 Race detector

#### 6.4.1 Constructing the Happens-Before graph

The race detector either reads in the events as they happen (online), or from an external text file. In both cases each event is deserialized from a JSON string into an internal representation. Each event is then added to hash tables so that it can be looked up by its various values (pid_out, mid_out, etc.). For each event in the execution trace, all of the applicable happens-before rules are checked and the appropriate edges added to the graph. Rule lookup is efficient as the events are added in trace order, the values pid_in and mid_in are guaranteed to be unique, and the happens-before is transitive. Thus evaluating a happens-before rule is a simple lookup in the appropriate hash table for each rule, and only the latest edge needs to be added. A sanity check ensures that all edges are valid.

#### 6.4.2 Race detection

The race detector as currently implemented is very simple: First, all combinations of events containing read or write operations on the same flow table are generated. Then, for each combination of events the race detector determines if they are ordered by doing a depth-first search of the happens-before graph. All read-write or write-write conflicts of concurrent events are marked as races.

#### 6.4.3 Reducing the number of reported races

Each of the races found by the race detector is then checked for commutativity using the commutativity specification. For added accuracy non-commutativity can optionally be verified by simulating both reorderings of the two switch operations. The remaining races are then optionally filtered to reduce the number of read-write races reported.

#### 6.4.4 Output

The race detector reports races in textual as well as graphical form. After the race detector is run, a Graphviz compatible file is generated that includes a graphical representation of the happens-before graph as well as a visual representation of the commuting and non-commuting races found by the race detector.
Chapter 7

Evaluation

We successfully used the analyzer to find concurrency violations in the synthetic examples listed in §2, as well as real-world applications based on the POX and Floodlight platforms.

7.1 Finding concurrency violations

7.1.1 Stateful firewall

We first run the stateful firewall application implemented as a Floodlight module as shown previously in Fig. 2.1. We send a packet from Host 1 to Host 2, and observe 3 high level concurrency violations: One is between 10–11, while the other two are between 8–10 and 8–11. Out of the reported three, all but one commute and are harmless: 10–11 commute as the two FLOW_MOD events match disjoint set of packets, and 8–10 commute as the rule in 10 applies only to packets originating from Host 1. The only potentially harmful race is between 8–11, i.e. between the FLOW_MOD installing the reverse flow rule and the reply packet sent from Host 2.

The controller can be fixed by inserting a BARRIER_REQUEST message after the two FLOW_MOD messages. This reduces the number of concurrency violations to a single one which commutes: the race between the two FLOW_MOD messages 10–11.

7.1.2 Load balancer

The load balancer application previously shown in Fig. 2.2 is implemented as a POX module. Running the application and sending a packet from Host 1 to the virtual IP address results in a single concurrency violation between events 6 and 8, corresponding to the packet sent and the FLOW_MOD message being installed on S2 respectively. For most executions the usual case is that the packet arrives (6) well after the rule is installed (8), which may lead to the belief that the controller is working correctly when it is actually not. SDNRacer will uncover such issues even if the problematic ordering does not appear as such in the trace. For this scenario, SDNRacer reports 1 harmful race between the read event when the packet arrives on the second switch (6) and the write of the new rule on S2, indicating that there is an issue.

Fixing the controller involves sending a BARRIER_REQUEST message to each switch after each FLOW_MOD message. Then, the switch must wait until it has received a BARRIER_REPLY from both switches before sending the PACKET_OUT message. Note that the number of reported races depends on how the wait for the BARRIER_REPLY message is implemented in the controller. If the controller simply waits in the current event handler, then the controller instrumentation as it is currently implemented does not know that there is an happens-before ordering between the FLOW_MOD 6 and the PACKET_OUT 4. This will result in a race being reported. With the current instrumentation it is necessary that the controller reacts to the BARRIER_REPLY message being received from S2.
7.2 Performance of the analyzer

In addition to running the analyzer on small sample programs, we also ran a stress test using the POX 12_multi controller. The controller was exercised for 10 minutes using the STS Fuzzer component, which generates random network events such as disconnecting and connecting switches, sending random packets such as ICMP pings and ARP requests, as well as randomly dropping packets throughout the network. A 32-switch fully connected mesh was chosen as the topology.

<table>
<thead>
<tr>
<th>Events</th>
<th>Table ops</th>
<th>Harmful r.</th>
<th>Commuting r.</th>
<th>Filtered r.</th>
<th>HB constr.</th>
<th>Race det.</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>101</td>
<td>243</td>
<td>321</td>
<td>643</td>
<td>0.3s</td>
<td>0.4s</td>
</tr>
<tr>
<td>1000</td>
<td>216</td>
<td>922</td>
<td>1900</td>
<td>2245</td>
<td>0.7s</td>
<td>2.8s</td>
</tr>
<tr>
<td>2000</td>
<td>636</td>
<td>1324</td>
<td>2702</td>
<td>18539</td>
<td>1.6s</td>
<td>6.8s</td>
</tr>
<tr>
<td>4000</td>
<td>1583</td>
<td>1324</td>
<td>2702</td>
<td>57397</td>
<td>3.7s</td>
<td>12.4s</td>
</tr>
</tbody>
</table>

Table 7.1: Performance of the race detector for various trace sizes

All tests were run on a 2.2 GHz i7-2720QM CPU and 16GB or RAM. The analyzer is currently implemented in Python and is not specifically optimized for performance. A substantial improvement in runtime is possible by making use of fast algorithms for race detection such as the ones described in [14]. Note that both the commutativity specification as well as filtering significantly reduce the number of reported races.
Chapter 8

Related Work

Several research projects are aimed at verifying the correctness of SDN networks. Tools such as HSA [10] and Libra [15] take snapshots of the network forwarding state and check if they violate certain properties. Tools such as Veriflow [16], NetPlumber [17], and Anteater [18] advance this by monitoring changes to the network in realtime, making it possible to check each rule update against a network given policy. These tools can detect interesting invariant violations, however beyond the specific rule violating the invariant they cannot tell what sequence of events led to them.

OFRewind [19] enables manual debugging of OpenFlow networks by running as a proxy between OpenFlow switches and controllers. It can capture both control and data plane traffic, and can re-inject traces into the network. Similar efforts are made with ndb [13], a network debugger that sits between the controller and switches and enables tracing of packets across the network. These tools are invaluable for debugging, however they do not automatically explore different scenarios.

STS [12] simulates a complete network topology in order to automate network debugging and root cause analysis for different network topologies and OpenFlow controllers. However, STS does not use a formal specification of partial orderings between events or the conditions under which two operations commute. Instead, it aims to enforce determinism and a global ordering as a means of generating reproducible traces suitable for delta debugging.

Model-checking approaches such as NICE [20], Kaui [21] and others [22–25] take a more formal approach and verify models of OpenFlow controllers for their correctness. Models are either extracted from existing controllers or written in languages suited for model checking. Unfortunately, these techniques suffer from the state-explosion problem which can make them hard to use in practice. In contrast, our analyzer is fast and even though it cannot guarantee complete controller correctness it is effective in detecting concurrency issues.

A separate group are the domain-specific languages such as the Frenetic language [26], the FatTire language [27], and NetCore [28] which aim to eradicate bugs in OpenFlow networks altogether. This is achieved by adding layers of abstraction and forcing developer to use higher-level constructs or declarative languages instead. These are then executed by a run-time system or compiled by a compiler that ensures correctness. Developers using such systems can use more abstract thinking, but must trust that the run-time system being used is sound.

OpenFlow is asynchronous, and there is research on how to provide consistency guarantees under certain circumstances, e.g. how to ensure per-packet or per-flow consistency during policy changes [29]. The authors of Attendre [30] propose a mechanism using versioned flow table entries and packet buffers that would eliminate certain classes of race conditions in OpenFlow networks. The main problem with these solutions is that they do not scale: as flow tables and buffers are always of finite size, solutions relying on them cannot make any guarantees for the worst case when the network is congested. However, they do help in situations where there is guaranteed to be enough space available for a given task.
Chapter 9

Conclusion

We introduced a happens-before relation for OpenFlow networks capturing the ordering between concurrent events. We also introduced a commutativity specification of the switch enabling precise analysis of the network. Based on these two ingredients, we developed a dynamic analyzer able to detect concurrency violations occurring between an OpenFlow controller, the underlying network switches and the packets in a given network. The analyzer works and is able to detect races in applications, including harmful ones capable of causing anomalies such as loss of reachability. The analyzer is furthermore able to filter out races that are most likely not of interest to the user and works quickly.

Future work
In future work we plan to: i) extend state aware commutativity checks to other operations, ii) perform more complex analysis of the controller to add happens-before orderings beyond the currently supported reactive mode, and iii) incorporate model checking techniques to address the inherent state-explosion problem. Furthermore we would like to increase the performance of the analyzer by employing more sophisticated algorithms for race detection.
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):
Detecting Concurrency Violations in Software-Defined Networks

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
Miserez

First name(s):
Jérémie Raymond

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich, April 20, 2015

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.