SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

June 2019, Vol. 62 062101:1–062101:20
https://doi.org/10.1007/s11432-018-9826-x

Predictive analysis for race detection in software-defined networks

Gongzheng LU[1,2], Lei XU[1*], Yibiao YANG[1] & Baowen XU[1*]

[1]*State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing* 210023*, China;*
[2]*School of Computer Engineering, Suzhou Vocational University, Suzhou* 215104*, China*

Received 31 July 2018/Revised 13 November 2018/Accepted 18 February 2019/Published online 8 May 2019

Abstract Race condition remains one kind of the most common concurrency bugs in software-defined networks (SDNs). The race conditions can be exploited to lead to security and reliability risks. However, the race conditions are notoriously difficult to detect. The existing race detectors for SDNs have limited detection capability. They can only detect the races in the original traces (observed traces) and cause false negatives. In this study, we present a predictive analysis framework called SDN-predict for race detection in SDNs. By encoding the order between the specified network events in SDNs as constraint, we formulate race detection as a constraint solving problem. In addition to detecting the races in the original trace, our framework can also detect the races in the feasible traces got from reordering the events in the original trace while satisfying the consistency requirements of trace. Moreover, we formally prove that our predictive analysis framework is sound and can achieve the maximal possible detection capability for any sound dynamic race detector with respect to the same trace. We evaluate our framework on a set of traces collected from three SDN controllers (POX, Floodlight, ONOS), running 5 representative applications including reactive and proactive applications in large networks, on three different network topologies. These experiments show that our framework has higher race detection capability than exisiting SDN race detector-SDNRacer, and detects more 1173 races. These 1173 races were previously undetected and confirmed by checking the race graphs.

Keywords constraint solving, predictive analysis, race detection, software-defined networks

Citation Lu G Z, Xu L, Yang Y B, et al. Predictive analysis for race detection in software-defined networks. Sci China Inf Sci, 2019, 62(6): 062101, https://doi.org/10.1007/s11432-018-9826-x

1 Introduction

Software-defined networks (SDNs) is a new network paradigm that facilitates network management and enables programmatical network configuration. SDN consists of control panel and data panel. The control panel of the network makes decision about how packets should flow through, and how the data panel of the network forwards packets according to the decisions made by control panel. SDN separates the control panel from the data panel. SDN controllers realize the control logic based on controllers' south bound interface, usually OpenFlow [1], to compute, maintain, populate the forwarding flow table of each SDN switch in the network.

The SDN controller is the brain of the whole network, and developing reliable control softwares in the asynchronous and distributed environment is vital. However, developing such highly asynchronous softwares is difficult due to the asychronism that results in concurrency bugs [2,3] such as race conditions (race and race condition will be alternately used in this paper). Owing to the packet forwarding and

* Corresponding author (email: xlei@nju.edu.cn, bwxu@nju.edu.cn)

message sending are frequently in SDN, the data packets and OpenFlow messages will exhibit a large number of race conditions.

In the context of SDN, the concurrency bugs can occur at two places: (i) within the control panel (the control software is multithreaded or distributed); (ii) at the interface between the control panel and the data panel (two events concurrently access to the same flow table of the switch, and one of the event is a write produced by the controller). The flow table of the SDN switch is similar to the memory location which is read and modified by several events and entities. The concurrency bugs in SDNs are difficult to detect, since these are only manifested in specific sequences of the events. However, detecting these bugs is important as they can cause packet loss, or increase packet forwarding delay and processing overhead on switches and the controller. These problems greatly lower the performance of the SDN. The first kind of concurrency bug has been researched by Xu et al. [4] using the happen-before (HB) causality model of the SDN controller. The second kind of concurrency bug can be detected using the dynamic analyzer-SDNRacer, which also uses the HB relations [5] between the events. The HB based approaches have limited detection ability and result in false negatives [6], since the HB based approaches can only detect the races in the observed traces by checking whether there is path between two concurrent events in the HB graph, which is a graph to specify the HB relations between events. In the HB graph, nodes denote events and directed edges represent the HB relations between the source nodes and the destination nodes. Two concurrent events form a race if there is no path between them, namely there is no HB relation between them. If we reorder the events in the observed trace under some contraints, two concurrent events between which there is HB relation in the observed trace may form a race in the reordering traces. The HB based approaches cannot find such races.

Predictive trace analysis [6–9] is a powerful technique to detect concurrency bugs, and has attracted attention in recent years. Predictive trace analysis can generate other permutations of the events in the trace under certain constraints and detect the bugs undetected in the original trace. Unlike dynamic analysis, it is capable of exposing the bugs in unexecuted traces, so it has higher detection capability. And it incurs fewer false positives than static analysis which uses whole program information.

Our work. In this paper, we present a predictive analysis framework called SDN-predict for race detection in SDNs. First, We encode the order between the specified network events as first logic formula while satisfying the consistency requirements of trace. Then, we use a SMT solver Z3[1)] to solve these formulas, and decide whether two concurrent access events exist race according to the satisfiabiltiy of the formula. Finally, we use the commutativity and time filters in [10] to filter the harmless races. We evaluate our framework by analyzing five applications on real world SDN controllers, and show that it discovers several undetected race conditions and has higher detection ability than existing SDN race dectection technique SDNRacer, which is a dynamic analysis approach based on HB model.

Contributions. The main contributions of this paper are summarized as follows.

(1) We present a predictive analysis framework SDN-predict for race detection in SDNs, and formulate race detection as constraint solving [11] problem. We give the definition of the consistency of trace especially for SDN considering the asynchronous nondeterminism, and prove the soundness and maximality of our predictive analysis framework.

(2) We briefly introduce an implementation of our framework to detect the races between the concurrent accesses of the flow tables.

(3) We evaluate our framework on five real world SDN applications, and compare it with SDNRacer. The experimental results shown that it has higher race detection capability than SDNRacer and detects 1173 previous undetected race conditions, and it has better scalability in the most cases.

The rest of this paper is organized as follows. Section 2 introduces the background. Section 3 presents our predictive race analysis framework SDN-predict. Section 4 briefly introduces the implementation of our framework. Section 5 evaluates SDN-predict. Section 6 reviews related work and Section 7 draws a conclusion of this paper.

1) SMT solver Z3. https://github.com/Z3Prover/z3.

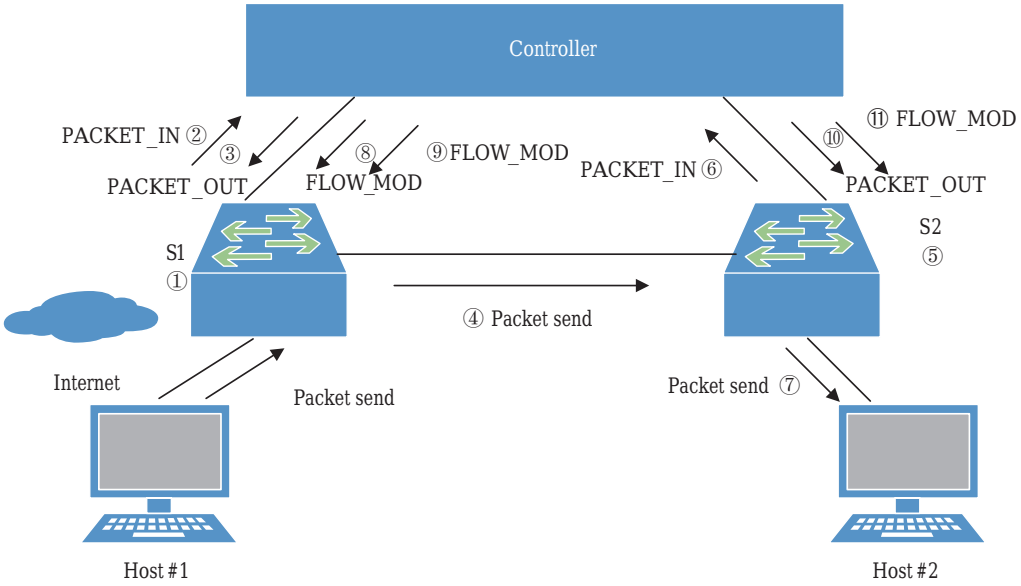

Figure 1 (Color online) An example of a LearningSwitch application in Floodlight and a sequence of events (represented by numbers) which causes three race conditions.

2 Background

In this section, we start with some background notions on concurrency problems in SDN programming firstly. Then, we give a motivating example to illustrate how concurrency bugs can occur in SDNs.

2.1 SDN programming and concurrency problems

A SDN controller is an event-driven programmable software application that acts as a stragetic control point in the SDNs. It is used to compute, maintain and populate the forwarding flow table of each SDN switch in the network. The forwarding flow table includes a list of flow table entries ordered by their priorities. Furthermore, each flow table entry is composed of a boolean predicate and a forwarding action. The flow table entry is used to match the packets transferred in the network. The predicate can identify the packets to which the forwarding action is applied. The forwarding actions are actions which can be taken when a packet matches the terms of a flow table entry. The supported forwarding actions include sending the packet to the controller for processing, sending the packet out to a specified port, flooding and forwarding using nonOpenFlow methods such as "normal" switch processing.

The messages emitted by asynchronous events, such as PACKET_IN showing the packet arriving at a switch and FLOW_REMOVED informing the controller that flow has been removed, can be dispatched to the controller nondeterministically. The order of these events may change in another execution. The nondeterminism in general is root cause of concurrency problems. We say that a race occurs when two events access to the same flow table at the same time, and at least one of the event is a write produced by the controller.

2.2 Motivating example: Floodlight LearningSwitch

Here, we take a Floodlight program which runs a LearningSwitch application [12] as example (see Figure 1). In this application, the switch will examine each packet and learn the source-port mapping. Thereafter, the source MAC address will be associated with the port. If the destination of the packet is already associated with some ports, the packet will be sent to the given port, otherwise it will be flooded on all ports of the switch.

Now, considering the following sequence of events: a host, Host#1, sends a packet to another host, Host#2, in the network. The packet hits the switch S1 in the network. S1 receives the packet and processes it ①. The packet is matched against the flow table in S1 to determine the flow table entry that

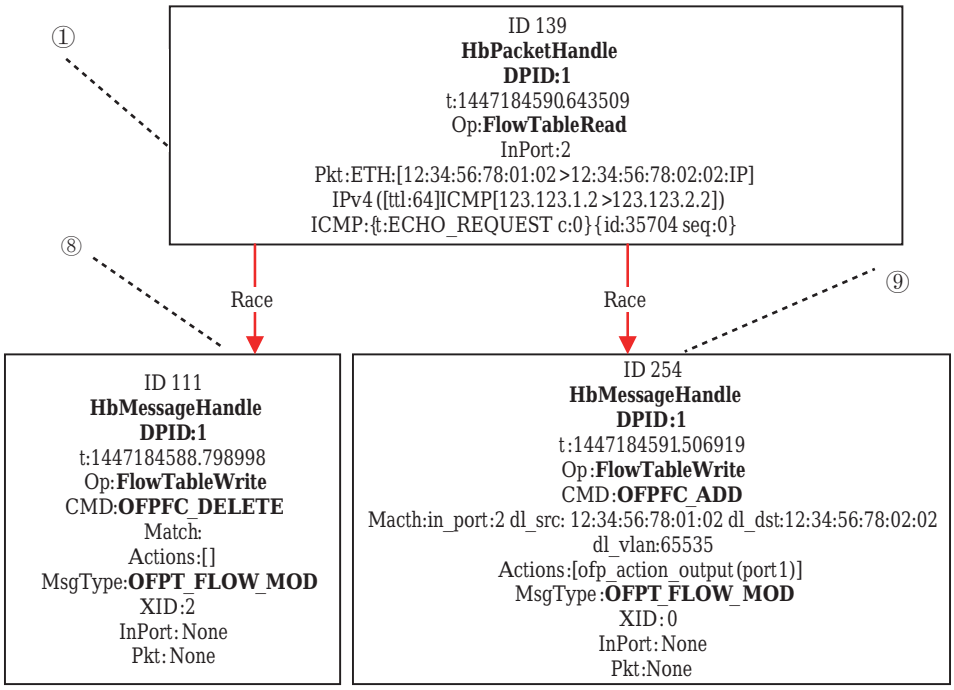

Figure 2 (Color online) A segment of event sequence in Figure 1 and two race conditions detected by SDN-predict.

should be applied. If there is a matched flow table entry, then S1 sends the packet out to other switches or to another host. Otherwise, S1 sends a message PACKET_IN to the controller and stores the packet in its buffer ②. When the controller receives the message, it (i) computes the shortest path between S1 and Host#2; (ii) sends the request back to S1 in a PACKET_OUT message ③; (iii) sends a Flow_MOD message containing delete operation ⑧ (deleting the flow entry in the specified flow table, this case may occur before the packet matching against the flow table in S1 if the flow entry is time out) and another Flow_MOD message containing add operation ⑨ (adding a new flow entry to the specified flow table). Then S1 reads a packet from its buffer and forwards the packet to specified switch S2 ④. S2 handles the packet similar to S1. It receives the packet sent by S1 and handles it ⑤. In term of whether the packet can match the flow table entry in its flow table, it either forwards the packet to Host#2 ⑦ or sends a message to the controller ⑥, and then, the controller sends the request back to S2 in a PACKETR_OUT message ⑩ and sends a FLOW_MOD message ⑪. In order to express the sequence of events conveniently, we use the number without circle to express the serial number of the event in later.

Considering the sequence of events 1-2-3-4-8-9-5-6-10-11-7. SDNRacer can only detect the race condition $(1, 8)$, because there is no HB relation between the two events. Figure 2 presents that when this race condition occurs, the matched flow table entry in S1 has been deleted before the packet match the flow table entry in the flow table, so the packet is dropped. The reason for this result is that the packet cannot match any flow table entry in S1 before the flow table entry added into the required flow table. Except for the race condition $(1, 8)$, SDN-predict detected another two race conditions $(1, 9)$ and $(5, 11)$ which are shown in Figures 2 and 3, respectively. The reason for the two races can be detected by SDN-predict is that, events 2 and 6 are asynchronous events which can happen nondterministically (In fact, when events 9 and 11 arrived at the switches, the packet matching process of events 1 and 5 have not finished, so two pairs of the concurrent events may access the flow table at the same time respectively), and predictive analysis can reorder them in the event sequence. The sequence 1-2-3-4-8-9-5-6-10-11-7 can be reordered into 2-3-8-9-1-4-5-6-10-11-7 and 1-2-3-4-8-9-6-10-11-5-7 as long as the reordered event sequences (traces) are feasible. When the race condition $(1, 9)$ occurs, S1 receives a packet and then reads the flow table to match the packet with the flow table entry while the controller sends an OpenFlow message FLOW_MOD to S1 to modify the flow table. It leads to packet loss. Race $(5, 11)$ causes the same result as $(1, 9)$. Next section we will describe our framework in detail.

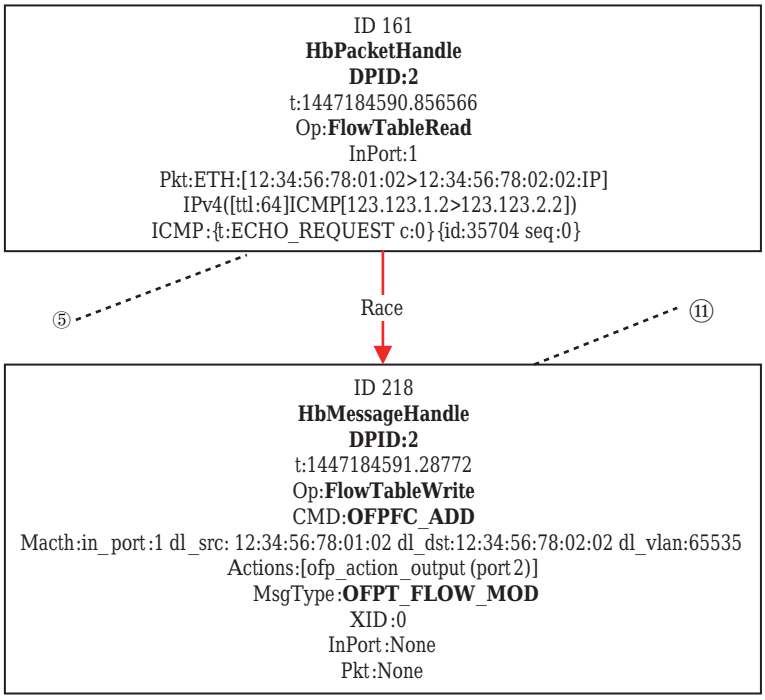

Figure 3 (Color online) Another segment of event sequence in Figure 1 and a race condition detected by SDN-predict.

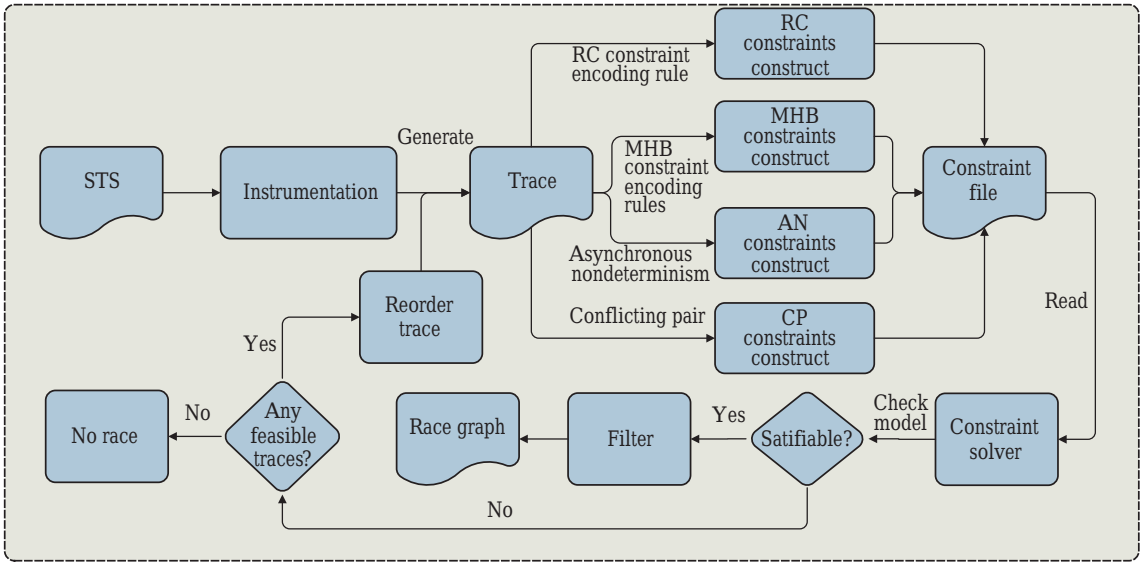

Figure 4 (Color online) The overview of SDN-predict.

3 Predictive analysis framework SDN-predict for SDNs

Firstly, the overview of the predictive analysis framework proposed in this paper is introduced briefly. Then, the events occurring in the network and the trace consisted of these events are presented. Next, the flow table in the OpenFlow switch [1] accessed by the events in the trace is given. In the last of this section, we describe our predictive analysis framework in detail.

3.1 The overview of SDN-predict

As shown in Figure 4, The STS (SDN troubleshooting simulator) simulates a complete network and the controller, and it should be instrumented firstly to log the packets, messages and operations in the

Table 1 The events in each SDN device and their related messages and operations

Device	Event	Messages	Operations
Switch	SwPktHandle	PACKET_IN	Read
	SwMsgHandle	PACKET_OUT, FLOW_MOD	Read, add, modify, delete
	SwPktSend	$\perp$	$\emptyset$
	SwMsgSend	PACKET_IN, FLOW_REMOVED	$\emptyset$
	SwFlowRemoved	FLOW_REMOVED	Delete
Controller	CtrlMsgHandle	$\perp$	$\emptyset$
	CtrlMsgSend	PACKET_OUT, FLOW_MOD	$\emptyset$
Host	HostPktHandle	$\perp$	$\emptyset$
	HostPktSend	$\perp$	$\emptyset$

switches and hosts. The logged information is written into trace files. Then we read each trace file and construct four kind of constraints for the events in the trace using the constraint encoding rules presented in this section. After that, the constraint solver reads the constraint file and checks its satisfiability. If the constraint file is satisfiable then a race condition occurs. The detected race conditions may be harmless and cause false positives. Thence SDN-predict uses the filtering policies in [10] to filter the harmless race conditions. At last, each of the remaining race condition is separately stored in a dot file, in which a race graph gives an event sequence causing the race condition. Otherwise the constraint file is unsatisfiable, SDN-predict uses the constraint solver to check whether there are feasible traces of the observed trace. If there are feasible traces, SDN-predict reorders the observed trace, and repeats above procedure until no feasible trace exists. If all the feasible traces are unsatisfiable, SDN-predict reports no race.

3.2 Events and trace

3.2.1 *Events*

Each event has a set of attributes which describes the event. The set of attributes is usually a subset of (sw, pid, mid, out_pids, out_mids, msg_type, ops) depending on the event type. The meaning of each attribute is as follows. sw is a switch identifier which receives or sends the event. pid is the identifier of the packet processed by the event. mid specifies the identifier of the OpenFlow message processed by the event. If there is no packet or message processed by the event, these attributes of the event can be set to the undefined value $\perp$. out_pids represents the identifiers of the packets sent out by the event. The packet sent out by the event is treated as different from the packet before being processed by the event, so the packet has a new unique pid in out_pids. An event may send out multiple packets, thus out_pids is a set. out_mids is the identifiers of the OpenFlow messages forwarded by the event. out_mids is also a set. If there is no such packets or messages, out_pids or out_mids sets to $\emptyset$. The type msg_type of OpenFlow messages processed by the SDN events in our framework includes: PACKET_IN, PACKET_OUT, BARRIER, FLOW_REMOVED, FLOW_MOD, PORT_MOD. ops is the set of flow table operations the event performed.

The events involved in this paper and their related messages and operations are listed in Table 1. Next, we will explain them in detail.

The events related to switch comprise:

SwPktHandle(sw, pid, out_pids, mid,out_mids, ops). On receipt of a data plane packet pid, the packet is matched against the flow entries of the flow table to select one. There are two cases during the matching process: (1) if a flow entry is founded, the instruction set in the flow entry is executed, the packet may be directed to another flow table or forwarded to another switch/host; (2) if the packet does not match a flow entry in the flow table, this is a table miss, and the unmatched packet may be dropped or passed to another table or sent to the controller via a PACKET_IN message.

SwMsgHandle(sw, pid, out_pids, mid,out_mids, msg_type, ops). On receipt of a message mid with message type msg_type, the switch may read a packet pid from its buffer and process the flow entries matched with the packet using the operation ops in the event sent by controller. After then, OpenFlow

messages can be sent to the controller (in which case out_mids contains the messages to be sent), and a packet can be forwarded (in which case out_pids contains the packet to be forwarded).

SwPktSend(sw, pid, out_pids). The switch sw sends the packet pid with a new identifier in out_pids out to another switch or host as a result of a SwPktHandle or SwMsgHandle event.

SwMsgSend(sw, mid, out_mids). The switch sends an OpenFlow message mid to the controller with the identifier in out_mids. For example, when a packet is received at the switch, and the packet cannot be matched against any flow entry of the flow tables in the switch, it sends a PACKET_IN message to the controller. As well as when a flow entry is removed by the expiry mechanism or the controller, an OpenFlow message FLOW_REMOVED is sent to the controller.

SwFlowRemoved(sw, mid, out_mids, ops). A flow entry in the switch is timed out or was explicitly deleted. As a result of this event, the switch generates a FLOW_REMOVED message to inform the controller (in which case the out_mids contains the message to be sent) that the flow entry is removed. ops is the operation delete in the FLOW_MOD message sent by the controller.

The events related to controller include:

CtrlMsgHandle(mid, out_mids). The controller receives and processes the OpenFlow message mid from the switch, and generates OpenFlow messages in out_mids as a response.

CtrlMsgSend(mid, out_mids). The controller sends the OpenFlow message mid such as a PACKET_OUT message out with the identifier in out_mids.

The events related to host involve:

HostPktHandle(pid, out_pids). The host receives and processes the packet pid, and generates packets in out_pids.

HostPktSend(pid, out_pids). The host sends the packet pid with identifier in out_pids out to another host or to a switch.

3.2.2 *Trace and consistent trace*

An execution trace τ is abstracted as a sequence of events. Given a trace τ and a concurrent object o (in this paper is flow table), $\tau\restriction_o$ is the restriction of τ to events involving o. $\tau\restriction_{\text{read}}$ is the projection of τ to the events containing read operation. If e is an event in trace τ, then let τ_e denote the prefix of τ up to and including e. For example, there is a trace $\tau = \tau_1 e \tau_2$, the prefix τ_e of τ is $\tau_1 e$. Let $\text{last}_{\text{op}}(\tau)$ be the last event of τ containing operation op, such as, $\text{last}_{\text{add}}(\tau)$ is the last event of τ which contains an add operation.

A trace τ is consistent which means that it satisfies the following conditions.

• **Read consistency.** An event containing a read operation (called read event) matches the packet against the flow entries of the flow table written by the most recent event containing a write operation (called write event). Formally, if e is a read event of τ, then $\text{ft}(e) = \text{ft}(\text{last}_{\text{write}}(\tau_e\restriction_{\text{ft}}))$, ft is the flow table e operated on.

• **Must happen-before.** Must happen-before (MHB) relation $\prec \subseteq \text{Event} \times \text{Event}$ represents the order between the events in a given trace τ. If event e_1 must happen-before event e_2, we can write as $e_1 \prec e_2$. For a finite trace consisting of a sequence of events $\tau = e_1 \cdot e_2 \cdots e_n$, we use $e_i \prec_\tau e_j$ to denote that event e_i must happen-before event e_j in τ. The MHB relations in the SDN are constructed by making use of the information provided by the attributes pid, out_pids, mid and out_mids of the events. For example, e_1 is a SwPktHandle or SwMsgHandle event, e_2 is a SwPktSend event, and $e_2.\text{pid} \in e_1.\text{out\_pids}$, then e_1 must happen-before e_2. All MHB relations in the SDN and their detailed explanations can reference to the MHB constraints in Subsection 3.4.2. In our example, $1 \prec 2$, $2 \prec 3$, $3 \prec 4$, $2 \prec 9$, $4 \prec 5$, $5 \prec 6$, $6 \prec 10$, $6 \prec 11$, $10 \prec 7$.

• **Asynchronous nondeterminism.** Asynchronous nondeterminism indicates that the asynchronous event (such as the event emitting a PACKET_IN message) is able to change its order with the read event on the same switch in another execution meanwhile keeping the MHB relations between the events except for the asynchronous event and the read event in the original trace. Formally, if there is a trace $\tau = \cdots e_i \cdot e_j \cdots$, where e_i is a read event and e_j is an asynchronous event, and $e_i \prec e_j$ in the original

trace, then it is able to $e_j \prec e_i$ in another execution while keeping the remaining MHB relations in the original trace. In our example, the MHB relation $1 \prec 2$ in the original trace can be $2 \prec 1$ in another trace due to asynchronous nondeterminism.

The above three conditions state the consistency of a trace. Consistency is just used to describe the property of a single trace. Any trace produced by the SDN applications is expected to consistent. However, various traces generated by the SDN applications are related. This means that some traces can be generated from a trace by reordering the events in this trace.

Let feasible(τ) be the set of all traces generated from the original trace τ, which we call feasible traces. The most common character of feasible(τ) is to require feasible(τ) to closed under consistency. However, this requirement is too strong. We use the weaker requirement: prefix closed [6], strict consistent [8] to give predictive analysis technique the largest coverage. Prefix closed says that prefixes of a feasible trace also is feasible. That is, if $\tau_1\tau_2 \in$ feasible(τ) then $\tau_1 \in$ feasible(τ). Strict consistent means that the flow table read by the event in feasible(τ) can be different from the one read by the same event in τ, but it should be consistent with the flow table written by the lastest write event. Formally, if $\tau_1 e_1, \tau_2 \in$ feasible(τ) and $\tau_1 = \tau_2$, then (1) if $\tau_2 e_1$ is consistent then $\tau_2 e_1 \in$ feasible(τ), and (2) if e_1 is a read event, then there exists some event e_2 with $e_2[\text{ft}(e_1)/\text{ft}] = e_1$ and $\tau_2 e_2$ is consistent, then $\tau_2 e_2 \in$ feasible(τ). A trace in feasible(τ) is called τ-feasible. In our example, the original trace τ: 1-2-3-4-8-9-5-6-10-11-7 is feasible according to the definition of the consistency. Event 1 read the flow table last written by most recent write event (initial write event), event 5 read the flow table last written by most recent write event (initial write event). And the events in trace τ all satisfy the MHB relations in SDN. Let's consider the trace τ_1: 2-3-8-9-1-4-6-10-11-5-7 generated from τ by reordering the asynchronous events 2 and 6. In τ_1, event 1 read the flow table written by the most recent event 9, and event 5 read the flow table written by the most recent event 11 (strict consistent), and all the events satisfy the MHB relation in SDN except for the asynchronous events, so according to the consistency of the trace, τ_1 is feasible, that is, $\tau_1 \in$ feasible(τ). The prefix 2-3-8-9-1-4 of τ_1 is consistent, so it is feasible. And, the flow table read by events 1 and 5 in τ_1 is respectively different from the one read by the same event in τ. Thus it can be seen that τ_1 meets the prefix closed, strict consistent.

3.3 Flow table

3.3.1 *The definition of flow table*

A switch is required to have at least one flow table, and can optionally have more flow tables. When the switch receives a packet, the packet is matched against the flow entries of the flow table to select one.

A flow table consists of flow entries, which further contains following components: match fields, priority, counters, instructions, timeouts, cookie, and flags. Match fields are used to match against packets. They consist of ingress port, packet headers and other fields. The match between a packet and a flow entry or between two flow entries can be either an exact match or a wildcard match. Priority is the matching precedence of the flow entry. If the packet matches to multiple flow entry, the flow entry with highest priority is selected. Counters represents the number of times the flow entry is matched. Instructions is used to modify the action set associated with the packet. Timeouts is the maximum amount of time or idle time before the flow entry is expired by the switch. Cookie is the opaque date chosen by the controller. Flags alter the way flow entries are managed.

A flow table entry fe is identified by its match fields (fe.m) and priority (fe.p): the match fields and the priority are used together to identify a unique flow entry in the flow table.

3.3.2 *Operations on flow table*

There are four types of operations that can be performed on the flow table: read, add, modify, and delete. The read operation is encapsulated by the SwPktHandle event and performed for each received packet. The add, modify, delete operations (we call them write operations) are encapsulated by the SwMsgHandle event with the msg_type of FLOW_MOD and performed when the OpenFlow message is processed. The race condition occurs when two events containing these operations access the same flow

table at the same time, and at least one of the event contains a write operation. We define the semantics of all above operations using the OpenFlow specification in the following:

read($\mathrm{pkt}, \mathrm{fe}_{\mathrm{read}}/\mathrm{fe}_{\mathrm{miss}}$). A read operation denotes that the switch receives a packet pkt, then the packet pkt is matched against the flow entry of the flow table. If there are such entries, the highest priority flow entry $\mathrm{fe}_{\mathrm{read}}$ is selected, otherwise the table-miss flow entry $\mathrm{fe}_{\mathrm{miss}}$ is selected.

add($\mathrm{fe}_{\mathrm{add}}$, OFPFF_CHECK_OVERLAP). Each add operation has an OFPFF_CHECK_OVERLAP flag which requires that the switch must first check for any overlapping flow entries in the requested table. Two flow entries fe_1 and fe_2 overlap if a single packet pkt may match both of them and the two entries have the same priority ($\mathrm{fe}_1.\mathrm{p} = \mathrm{fe}_2.\mathrm{p}$). If an overlap conflict exists between an existing flow entry fe and the flow entry $\mathrm{fe}_{\mathrm{add}}$ to be added by the add operation, the switch must refuse the addition. For non-overlapping add or those with non overlapping check, the switch must insert the flow entry $\mathrm{fe}_{\mathrm{add}}$ in the requested table. If there is a flow entry fe with the same match fields ($\mathrm{fe.m} = \mathrm{fe}_{\mathrm{add.m}}$) and priority ($\mathrm{fe.p} = \mathrm{fe}_{\mathrm{add.p}}$) in the flow table, then that entry fe must be cleared from the table and the new flow entry $\mathrm{fe}_{\mathrm{add}}$ will be added.

For the modify and delete operations, they have non-strict versions and strict versions. In the strict versions, all match fields and the priority are strictly matched against the flow entry, and only identical flow entry is modified and removed. In the non-strict versions, missing match fields are wildcarded and the priority is ignored.

We use $\mathrm{fe}_1 \overset{\text{strict}}{\equiv} \mathrm{fe}_2$ and $\mathrm{fe}_1 \overset{\text{non-strict}}{\equiv} \mathrm{fe}_2$ to respectively denote the strict and non-strict semantics of flow entry matching:

$$\mathrm{fe}_1 \overset{\text{strict}}{\equiv} \mathrm{fe}_2 := \mathrm{fe}_1.\mathrm{m} = \mathrm{fe}_2.\mathrm{m} \wedge \mathrm{fe}_1.\mathrm{p} = \mathrm{fe}_2.\mathrm{p} \wedge \mathrm{strict} = \mathrm{True},$$

$$\mathrm{fe}_1 \overset{\text{non-strict}}{\equiv} \mathrm{fe}_2 := \mathrm{fe}_1.\mathrm{m} \subseteq \mathrm{fe}_2.\mathrm{m} \wedge \mathrm{strict} = \mathrm{False}.$$

The boolean value of strict affects how the matching is performed.

modify($\mathrm{fe}_{\mathrm{mod}}$, strict). For the modify operation, if a matching entry fe that matches the flow entry $\mathrm{fe}_{\mathrm{mod}}$ existing in the table, the instructions field of this entry is replaced with the value from the operation. If no flow entry in the requestd flow table matches the operation, no table modifications occur.

delete($\mathrm{fe}_{\mathrm{del}}$, strict). For the delete operation, if a matching entry fe that matches the flow entry $\mathrm{fe}_{\mathrm{del}}$ existing in the table, it must be deleted, and if the entry has the OFPF_SEND_FLOW_REM flag, it should generate a FLOW_REMOVED message. If no flow entry in the requested flow table matches the operation, no flow table modifications occur.

3.4 Predictive analysis framework for SDN race detection

In this subsection, the formal definition of race condition is given firstly, then the basic idea of the predictive race detection is introduced simply and the core of our predictive analysis framework-how to encode the constraints used in the predictive analysis framework to detect the race conditions in SDNs is presented in the last.

3.4.1 *Predictive race detection*

Before introducing predictive race detection, we first give the formal definition of race condition.

Definition 1. Events a and b form a conflicting pair written $\mathrm{CP}(a, b)$, iff $\mathrm{op}(a) = \mathrm{write}$, $\mathrm{op}(b) \in \{\mathrm{write}, \mathrm{read}\}$, $\mathrm{ft}(a) = \mathrm{ft}(b)$, where $\mathrm{op}(a)$ is the operation event a contains.

Definition 2. Consistent trace τ has a race condition iff there is a consistent trace $\tau_1 ab \in \mathrm{feasible}(\tau)$ such that $\mathrm{CP}(a, b)$.

In our example, we consider the origianl trace τ: 1-2-3-4-8-9-5-6-10-11-7 again. There are four conflicting pairs: $\mathrm{CP}(1, 8)$, $\mathrm{CP}(1, 9)$, $\mathrm{CP}(8, 9)$, and $\mathrm{CP}(5, 11)$. Three of them are remaining race conditions. For conflicting pair $\mathrm{CP}(1, 8)$, there is a consistent trace τ_1: 2-3-9-8-1-4-5-6-10-11-7, $\tau_1 \in \mathrm{feasible}(\tau)$. For $\mathrm{CP}(1, 9)$, there is a consistent trace τ_2: 2-3-8-9-1-4-5-6-10-11-7, $\tau_2 \in \mathrm{feasible}(\tau)$. And for $\mathrm{CP}(5, 11)$, there

is a consistent trace τ_3: 1-2-3-4-8-9-6-10-11-5-7, $\tau_3 \in \text{feasible}(\tau)$. There is a consistent trace for $\text{CP}(8, 9)$, but it is filtered. Because the two events in this conflicting pair are for non-overlapping flow entries of the flow table, this race condition is harmless and is filtered using the commutativity filter in [10].

In generally, given an input trace τ, the goal of our predictive race detection is to find a τ-feasible trace τ' and a $\text{CP}(a, b)$ such that events a and b are next to each other in τ'. During the race detection, the particular flow table written or read by events are not relevant, in order to simplify the trace representation we make no distinction between the event that appears in original trace τ and its variants satisfying the strict consistent in τ-feasible traces.

We formulate the race detection problem for SDN as a constraint solving problem. In order to represent the contraints between the events, we introduce an order variable $X_{e.\text{eid}}$ for each event e in τ, which represents the order of event e in τ', where eid is the identifier of e. Then we generate a formula Φ over these order variables to represent the race detection problem for τ and $\text{CP}(a, b)$. Φ is satisfiable iff $X_{b.\text{eid}} - X_{a.\text{eid}} = 1$ for a certain trace $\tau' \in \text{feasible}(\tau)$. We can solve Φ using any constraint solver. If Φ is satisfiable, which means that there is a model (trace) for the formula, then (a, b) is a race condition in τ'.

3.4.2 *Constraint encoding*

In SDN, the formula Φ consists of four types of constraints: (1) MHB constraints, (2) read consistency (RC) constraints, (3) asynchronous nondeterminsim (AN) constraints, and (4) conflicting pair (CP) constraints. Thus, Φ is constructed by a conjunction of four sub-formulae:

$$\Phi = \Phi_{\text{mhb}} \wedge \Phi_{\text{rc}} \wedge \Phi_{\text{an}} \wedge \Phi_{\text{cp}}.$$

MHB constraints (Φ_{mhb}). MHB yields an obvious partial order $\prec$ on the events of τ which must be respected by any τ-feasible trace. MHB can be easily denoted as constraints over the order variables. Initially, $\Phi_{\text{mhb}} \equiv$ true, then we conjunct it with a constraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ whenever e_1 occurs before e_2. The order constraints between the events except four rules (BARRIERPRE, BARRIERPOST, TIME1, TIME2) can be constructed by making use of the information provided by the attributes pid, out_pids, mid and out_mids of the events. BARRIERPRE, BARRIERPOST describe the effect of the BARRIER request messages on the switch. TIME1 and TIME2 are used to represent the order constraints between the events which cannot be reordered because the time distance between these events exceeds the time window δ. The encoding rules for MHB constraints are listed in Table 2. The rules are described in the following.

Rules1–4 (pid_out $\rightarrow$ pid_in). These rules order the related packet processing events e_1 and e_2 using the informatin provided by their attributes pid and out_pids. If the pid of e_2 is contained in the out_pids of e_1, that is, two events process the same packet, then e_1 must occur before e_2. Such as Rule1, if e_1 is a SwPktHandle or SwMsgHandle event and e_2 is a SwPktSend event, and $e_2.\text{pid} \in e_1.\text{out\_pids}$, then the constraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ is get. Rules2–4 are similar with Rule1.

Rules5–8 (mid_out $\rightarrow$ mid_in). These rules order the related message processing events e_1 and e_2 using the information provided by their attributes mid and out_mids. If the mid of e_2 is contained in the out_mids of e_1, that is, two events process the same message, then e_1 must occur before e_2. In Rule5, if e_1 is SwPktHandle or SwMsgHandle or SwFlowRemoved and e_2 is SwMsgSend, and $e_2.\text{mid} \in e_1.\text{out\_mids}$, the constraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ is get. Rules6–8 are similar with Rule5.

Rules9, 10 (BARRIER). The switch can handle messages received from the controller in a different order from the one they were sent. To enforce the order between the events, the controller sends a BARRIER request message. Rule9 describes that all events before BARRIER must be processed before the BARRIER message. For example, if the msg_type of e_2 is BARRIER and e_1 occurs before e_2 in the trace τ, the constraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ is held. Rule10 describes that all events after BARRIER must be processed after the BARRIER message. The constraint can be encoded as in Rule9. If the msg_type of e_1 is BARRIER and e_1 occurs before e_2 in the trace τ, the constraint should be satisfied is $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$.

Rules11, 12 (ALT_BARRIER). These are the alternative form of Rules9 and 10. They order the event SwMsgHandle and the event triggering it (CtrlMsgHandle). Rule11 describes that if e_2 is a SwMsgHandle

Table 2 The encoding rules for MHB constraints ($X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$)

Category	Rule	Event 1	Event 2
pid_out → pid_in (e_2.pid ∈ e_1.out_pids)	1	$e_1 \in$ SwPktHandle ∪ SwMsgHandle	$e_2 \in$ SwPktSend
	2	$e_1 \in$ SwPktHandle ∪ SwMsgHandle	$e_2 \in$ SwMsgHandle
	3	$e_1 \in$ HostPktHandle	$e_2 \in$ HostPktSend
	4	$e_1 \in$ SwPktSend ∪ HostPktSend	$e_2 \in$ SwPktHandle ∪ HostPktHandle
mid_out → mid_in (e_2.mid ∈ e_1.out_mids)	5	$e_1 \in$ SwPktHandle ∪ SwMsgHandle ∪ SwFlowRemoved	$e_2 \in$ SwMsgSend
	6	$e_1 \in$ CtrlMsgHandle	$e_2 \in$ CtrlMsgSend
	7	$e_1 \in$ SwMsgSend	$e_2 \in$ CtrlMsgHandle
	8	$e_1 \in$ CtrlMsgSend	$e_2 \in$ SwMsgHandle
BARRIERPRE (e_2.msg_type = BARRIER e_1.sw = e_2.sw $e_1 \prec_\tau e_2$)	9	$e_1 \in$ SwMsgHandle	$e_2 \in$ SwMsgHandle
BARRIERPOST (e_1.msg_type =BARRIER e_1.sw = e_2.sw $e_1 \prec_\tau e_2$)	10	$e_1 \in$ SwMsgHandle	$e_2 \in$ SwMsgHandle
ALT_BARRIERPRE (e_2.msg_type = BARRIER e_1.sw = e_2.sw $e_1 \prec_\tau e_2$)	11	$e_1 \in$ CtrlMsgHandle	$e_2 \in$ SwMsgHandle
ALT_BARRIERPOST (e_1.msg_type = BARRIER e_1.sw = e_2.sw $e_1 \prec_\tau e_2$)	12	$e_1 \in$ CtrlMsgHandle	$e_2 \in$ SwMsgHandle
TIME1 ($e_2.t - e_1.t > \delta$)	13	$e_1 \in$ SwPktHandle ∪ SwMsgHandle	$e_2 \in$ SwMsgHandle
TIME2 ($e_2.t - e_1.t > \delta$)	14	$e_1 \in$ SwMsgHandle	$e_2 \in$ SwPktHandle ∪ SwMsgHandle
FlOW_REMOVED ($e_2.t - e_1.t > \delta$)	15	$e_1 \in$ SwMsgHandle	$e_2 \in$ AsyncFlowExpiry

event and its msg_type is BARRIER, and event e_1 is CtrlMsgHandle, then e_1 must happen-before e_2, the constraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ is held. And Rule12 says that if e_1 is CtrlMsgHandle and its msg_type is BARRIER, e_2 is SwMsgHandle, then e_1 must happen-before e_2. So the contraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ is constructed.

Rules13, 14 (TIME). These rules add constraints between events that are unlikely be reordered because the time distance δ between these events exceeds the time window, the value of δ can be inferred from related work. Rules13 and 14 describe the situations that if the time distance between the event that stores the packet in the switch buffer and the event retrieving the packet from the buffer is larger than δ, then the event with small time value occurs before the other one. In Rule13, e_1 is SwPktHandle or SwMsgHandle, e_2 is SwMsgHandle, and e_2 happens after e_1 and the time distance between them is larger than δ, the constraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ is constructed. Rule14 can be encoded similarly.

Rule15 (FLOW_REMOVED). This rule orders the event SwMsgHandle and the event AsyncFlowExpiry if the time distance between them is larger than δ. SwMsgHandle denotes that the switch reads or writes the flow entry when receives a PACKET_OUT or FLOW_MOD message from the controller. AsyncFlowExpiry denotes that the flow entry in the switch is timed out. In this rule, if e_1 is SwMsgHandle, e_2 is AsyncFlowExpiry, e_1 occurs before e_2 and the time distance between them is larger than δ, the constraint $X_{e_1.\text{eid}} < X_{e_2.\text{eid}}$ is held.

Read consistency constraints (Φ_{rc}). Recall the strict consistent [8] that the flow table read by the event in τ-feasible traces can be different from the one read by the same event in τ, but the flow table read by the event should be consistent with the one written by the latest write event in order to satisfy the read consistency requirement of τ-feasible traces. The last write event w must happen-before the corresponding read event r, and there is no other write events between w and r. In a trace τ, let read(ft)

1. MHB(Φ_{mhb}):
   $X_2 < X_3 \wedge X_2 < X_9 \wedge X_3 < X_4 \wedge X_4 < X_5$
   $\wedge\ X_6 < X_{10} \wedge X_6 < X_{11} \wedge X_{10} < X_7$;
2. Read consistency(Φ_{rc}):
   $(X_9 < X_1 \vee X_{w_0} < X_1) \wedge (X_{11} < X_5 \vee X_{w_0'} < X_5)$;
3. Asynchronous nondeterminism(Φ_{an}):
   $(X_1 < X_2 \vee X_2 < X_1) \wedge (X_5 < X_6 \vee X_6 < X_5)$;
4. CP(Φ_{cp}):
   CP$(1, 9)$: $X_9 - X_1 = 1$,
   CP$(8, 9)$: $X_9 - X_8 = 1$,
   CP$(5, 11)$: $X_{11} - X_5 = 1$,
   CP$(1, 8)$: $X_8 - X_1 = 1$.

Figure 5 The constraint encoding for the trace in our example.

be the read event r reading the flow table ft in the switch, and Write(ft) be the set of write events W writing the flow table ft, and Write(-\-) be the set of write events W' writing the flow table different from ft. For a read r, the read consistency constraints can be encoded as follows:

$$\Phi_{\text{rc}}(r) = \bigvee_{w \in W} \left(X_{w.\text{eid}} < X_{r.\text{eid}} \bigwedge_{w \neq w' \in W'} (X_{w'.\text{eid}} < X_{w.\text{eid}} \vee X_{r.\text{eid}} < X_{w'.\text{eid}}) \right).$$

Asynchronous nondeterminism constraints (Φ_{an}). For an asynchronous event e_j, it is able to change its order with a read event e_i on the same switch in another execution, so the asynchronous nondeterminism constraints can be written as $\Phi_{\text{an}} = X_{e_i.\text{eid}} < X_{e_j.\text{eid}} \vee X_{e_j.\text{eid}} < X_{e_i.\text{eid}}$.

Conflicting pair constraints (Φ_{cp}). For each conflicting pair CP(a, b), a constraint $X_{b.\text{eid}} - X_{a.\text{eid}} = 1$ is encoded for specified the race condition.

The constraint encoding of the trace τ: 1-2-3-4-8-9-5-6-10-11-7 in our example is shown in Figure 5 (here we use the serial number of the event instead of its eid), where w_0 and w_0' are the initial write events writing the flow table before the packets arrive at the switches. Based on the MHB relations between the events except for the asynchronous events in the example trace and the encoding rules for MHB constraints, the formula Φ_{mhb} is constructed. In following, events 9 and 11 are write events, and events 1 and 5 are read events, so the formula Φ_{rc} is in accord with the read consistency constraint. Events 2 and 6 are asynchronous events, they can change order with the read event on the same switch, thus Φ_{an} is obtained. Finally, there are four conflicting pairs, thus four CP formulas are get. Putting the first three constraints and each conflict pair constraint together, we invoke SMT solver Z3 to compute a solution for these order variables. For the four conflicting pairs, SMT sovler returns a solution for each, so they are all races, but $(8, 9)$ is filtered by the commutativity filter.

3.5 Soundness and maximality

Our predictive analysis technique above is sound and maximal. Soundness means every detected race is not false positive. Maximality means that our technique does not miss any race that can be detected by any sound race detected based on the same trace. In order to illustrate the soundness and maximality of our technique, we give Theorem 1.

Theorem 1. If Φ is the constraint encoding associated to a given trace τ, then Φ is satifiable iff (a, b) is a race in τ in Definition 2.

Proof. Suppose that $\tau = e_1 \cdot e_2 \cdots e_n$, and v is an valuation of order variables. Note that $v \vDash \Phi$ for some $v : \{X_{e_1}, X_{e_2}, \ldots, X_{e_n}\} \to \mathbb{N}$ iff there is a valuation $v : \{X_{e_1}, X_{e_2}, \ldots, X_{e_n}\} \to \{1, 2, \ldots, n\}$ satisfies the constraints in Φ. Because the particular values assigned to the X variables are irrelevant, except for the CP constraint $X_b - X_a = 1$, so we can find an ordering of $v(e_1), v(e_2), \ldots, v(e_n)$ such that $v(a)$ is followed by $v(b)$. Any v generates the permutation $e_{v(X_1)} \cdot e_{v(X_2)} \cdots e_{(X_n)}$ of τ, which we write $[v]$.

It is easy to see that $v \vDash \Phi_{\text{mhb}}$ iff $[v]$ satisfies the MHB consistency requirements, and that $v \vDash \Phi_{\text{an}}$ iff $[v]$ satisfies the asynchronous nondeterminism requirements. We can perform induction on i that for any

asynhronous event e_i of τ, it is the case that $v \vDash \Phi_{\mathrm{rc}(r)}$ iff $[v]_{e_i}\restriction_{\mathrm{read}} = \tau_{e_i}\restriction_{\mathrm{read}}$ and any read event r in these trace projections satisfies the read consistency requirement in $[v]_{e_i}$.

Let us first prove the soundness, that is, that if Φ is satisfiable then (a, b) is a race in τ. If $v \vDash \Phi$, that is, there is a valuation satifies the constraints, then by the properties above, we can get the following: $[v]$ satisfies the MHB and asynchronous nondeterminism consistency requirements; $[v]_b = [v]_a b$, and for all asynchronous events e_i, $[v]_{e_i}\restriction_{\mathrm{read}} = \tau_{e_i}\restriction_{\mathrm{read}}$. Then we can inductively construct a trace τ_1 over the set $\{e | e \prec a\}$ satisfying the prefix closed and strict consistent. We traverse these events in the order they occur in $[v]$, as follows, where e is the next such event: if e is not a read or an asynchronous event then append it to τ_1; if e is a read then to ensure the read consistency the value it read should be changed to the value written by the last write event τ_1 so far, and then append e to τ_1; if e is a write event, and there is a sequence $e_i \prec \cdots \prec e_j$, where e_i is an asynchronous event and e_j is last event in τ_1, and $e_j \prec e$, then the asynchronous event may change its order with the read event on the same switch, we can insert event in the sequence $e_i \cdots e_j \cdot e$ before the related read event meanwhile keeping the MHB relations between them and change the value of the read event to the value written by e. All the steps above preserve the consistency of τ_1 and satisfy the prefix closed and strict consistent characterizing feasible(τ), so we can conclude $\tau_1 \in$ feasible(τ). We can now extend τ_1 with a and b similar as above, and thus we get that $\tau_1 ab \in$ feasible(τ), so (a, b) is a race in τ.

Let us now prove the maximality, that is, that if (a, b) is a race in τ then Φ is satisfiable. Let $\tau_1 ab \in$ feasible(τ) and let τ_2 be the trace consists of the remaining events of τ, in the order which they appear in τ. Although $\tau' = \tau_1 ab \tau_2$ may be not τ-feasible, it still meet the MHB and asynchronous nondeterminism consistency requirements. Let v be the valuation with $[v] = \tau'$. Then it is easy to know $v \vDash \Phi_{\mathrm{mhb}} \wedge \Phi_{\mathrm{an}} \wedge X_b - X_a = 1$. Since $\tau_1 ab$ is τ-feasible, prefix closedness ensures that $[v]_e$ is also τ-feasible for all asynchronous event e; the strict consistent implies that $[v]_e\restriction_{\mathrm{read}} = \tau_e\restriction_{\mathrm{read}}$, so by the property above and the definition of read consistency we conclude that if a is a read event then $v \vDash \Phi_{\mathrm{rc}(a)}$, otherwise if b is a read event then $v \vDash \Phi_{\mathrm{rc}(b)}$, so $v \vDash \Phi$.

In this section, we describe our predictive analysis framework in detail, including the elementary concepts, the consistency and feasible of the trace which is our framework's base, and the constraint encoding procedure. In last, we prove the soundness and maximality of our framework. Next, the implementation of our framework will be presented.

4 Implementation

We implemented our predictive race analysis framework SDN-predict based on SDNRacer. The framework was implemented with Python. SDN-predict consists of two phases: instrumentation and predictive race analysis. During the instrumentation phase, SDN-predict instruments the STS which simulates a complete network and several controllers. STS intrumentation is used to log the packets, messages and operations in the switches and hosts. The logged traces are written into files. The controller instrumentation for POX[2)], Floodlight[3)], ONOS[4)] wraps the event handlers for the messages received from switches, and links the incoming messages received from switches with the corresponding outgoing messages sent to switches. Then the controller instrumentation passes this logged information to STS helping to filter harmless race conditions. We just used the instrumentaion policy in SDNRacer. During the predictive race analysis phase, SDN-predict reads the events from a trace file, and encodes the constraints among the events in the trace for each conflicting pair using the encoding rules described in Subsection 3.4.2. In the specified implementation, the conflicting pair constraint $X_b - X_a = 1$ for each CP(a, b) is simplified by replacing X_a with X_b in the constraints as in RVPredict[5)]. Sometimes, the contraints among the events in the trace form cycles denoting contradiction. The contraints are unsatisfiable if there is a contradiction. The solver cannot report the race conditions in these traces. We first check whether there are cycles in the

2) POX Repo. https://github.com/noxrepo/pox.
3) Floodlight Repo. https://github.com/floodlight/floodlight.
4) ONOS Repo. https://github.com/opennetworkinglab/onos.
5) RVPredict. http://fsl.cs.illinois.edu/rvpredict/.

contraints among the events, and then treat the events in the cycle as equivalent events. For example, if there are constraints $X_{e_1} < X_{e_2}$, $X_{e_2} < X_{e_3}$ and $X_{e_3} < X_{e_1}$, then these constraints form a cycle, we treat these three events in the cycle as equivalent events, and construct formula $X_{e_1} = X_{e_2}$ and $X_{e_2} = X_{e_3}$ and $X_{e_3} = X_{e_1}$. Later, SDN-predict uses constraint solver Z3 to solve the constraints. The default constraint solving time and memory are set to 50 s and 6000 MB for each conflicting pair. If the solver returns a solution, we report a race. Then the commutativity and time filters in [10] is used to filter out the harmless race conditions. The value of δ used in time filter is set to 2 s which is the best case proved in [10]. Each of the remaining race condition is separately stored in a dot file, in which a race graph gives an event sequence leading to the race.

5 Evaluation

In this section, we evaluate our predictive race analysis framework SDN-predict. Our evaluation aims to answer the following research questions:

(1) Race detection capability. How many races can our framework detect in representative SDN controllers, running different applications, on different network topologies? How many more races can it detect than SDNRacer (the unique SDN race detector on flow tables in switches)?

(2) Scalability. How efficient is our framework? Can it scale to large applications?

All experiments were conducted on a machine with 16-core 3.20 GHz Intel i5 processor and 4 GB memory under Ubuntu version 4.8.2. And we use SMT solver Z3 to solve the constraints. We discuss our experimental results in detail as presented in Table 3.

SDN controllers. We run SDN-predict on three controllers: Floodlight version 0.91, POX EEL, and ONOS version 1.2.2.

Applications. We select 5 applications including reactive and proactive applications in large networks and run them on three different topologies.

The selected applications include:

(1) LearningSwitch. It is a reactive application which builds and maintains a dynamic MAC address table for each switch. This table is a map between the MAC addresses and the physical port they can be reached on. We run this application on POX and Floodlight [12, 13].

(2) Forwarding. The LearningSwitch applications work on each switch so they are inefficiently. Most controllers include a forwarding application working on the whole network and building and maintaining a MAC addresses table throughout the network. We run this application on POX, ONOS and Floodlight [14–17].

(3) CircuitPusher. This is a proactive application which automatically installs paths between two hosts. We run this application on Floodlight [18].

(4) FireWall. This application allows or drops host communication based on a given operator policy. We run it on Floodlight [19].

(5) LoadBalancer. This application performs stateless load balancing among a set of replicas. When the controller receives the packet to be forwarded to the hosts which are identified by their virtual IP addresses, this application selects a host and installs the flow rules along the entire path. We run it on Floodlight [20].

Topologies. We run each controller on three different topologies — Star, Mesh, BinaryTree. The Star topology has one switch with two hosts connected. The Mesh topology has two switches with one host connected to each. The BinaryTree topology comprises seven switches in the form of a binary tree with each leaf switch connected to one host.

We used 30 traces collected from different applications, different controllers and different topologies using STS to evaluate our framework and compare its race detection capability with SDNRacer. Each trace gets from 200 simulation steps by STS. In each step, the host in the topology decided randomly whether to send a packet to another randomly selected host. Since LoadBalancer only makes sense with

Table 3 Reported races for different traces with applying time filter using $\delta = 2$ s[a)]

App.	Topology	Controller	Num. of events			SDN-predict				SDNRacer			
			Total	WR	RD	Races	Comm.	Time	Remain.	Races	Comm.	Time	Remain.
Learning-switch	Star	Pox EEL	193	7	42	309	255	74	**10**	294	218	66	**10**
		Floodlight	314	7	70	511	227	236	**48**	494	223	227	**44**
	Mesh	Pox EEL	274	16	66	570	403	143	**24**	532	387	121	**24**
		Floodlight	233	6	66	204	68	131	**5**	190	64	125	**1**
	BinTree	Pox EEL	4033	487	663	64086	62571	1449	**66**	62066	61337	664	**65**
		Floodlight	9230	1251	904	281380	274643	6076	**661**	275246	270207	4737	**302**
For-warding	Star	Pox Angler	106	4	16	68	35	25	**8**	61	33	21	**7**
		Pox EEL	145	8	19	180	135	35	**10**	109	84	24	**1**
		Pox EEL Fx	184	8	29	260	195	53	**12**	189	145	42	**2**
		ONOS	476	18	71	1431	1248	133	**50**	1336	1163	127	**46**
		Floodlight	97	3	13	42	14	18	**10**	35	13	14	**8**
	Mesh	Pox Angler	248	13	48	345	125	200	**20**	323	116	191	**16**
		Pox EEL	306	20	50	590	348	217	**25**	405	235	159	**7**
		Pox EEL Fx	303	16	51	464	348	97	**19**	276	206	62	**8**
		ONOS	880	44	181	4444	4135	256	**53**	4059	3781	228	**50**
		Floodlight	180	6	36	114	46	52	**16**	104	46	45	**13**
	BinTree	Pox Angler	2106	286	359	20935	13397	7230	**308**	20447	13179	6988	**280**
		Pox EEL	4362	504	453	51918	40203	11378	**337**	34385	27956	6201	**228**
		Pox EEL Fx	4283	467	413	44297	42977	1199	**121**	12509	12238	242	**29**
		ONOS	8031	1492	920	248521	244687	3472	**362**	236429	233578	2598	**253**
		Floodlight	1886	203	323	12681	11856	554	**271**	12293	11766	317	**210**
Circuit-pusher	Star	Floodlight	218	25	41	1325	1052	230	**43**	1301	1040	218	**43**
	Mesh	Floodlight	327	42	74	1974	1597	311	**66**	1933	1581	287	**65**
	BinTree	Floodlight	1200	144	227	6298	5610	639	**49**	6156	5605	507	**44**
FireWall	Star	Floodlight	190	3	36	111	37	65	**9**	104	35	62	**7**
	Mesh	Floodlight	221	6	48	150	58	76	**16**	139	56	69	**14**
	BinTree	Floodlight	841	52	170	1457	1093	278	**86**	1384	1090	228	**66**
Load-balancer	Star4	Floodlight	3889	822	476	728703	709379	17086	**2238**	703864	685158	16492	**2214**
	Mesh4	Floodlight	5475	1036	735	333483	327929	4796	**758**	319121	314091	4303	**727**
	BinTree	Floodlight	24612	6213	2163	4816843	4850370	64987	**1486**	4705379	4642118	62031	**1230**
Total races				–					**7187**		–		**6014**

a) The numbers in bold are the final numbers of races detected by SDN-predict and SDNRacer.

more than two hosts, so two traces of the LoadBalancer are collected from the larger topology (Star4 and Mesh4) which replaces two hosts with four.

As shown in Table 3, columns 4–6 report metrics of traces: total number of events (total), write events (WR), and read events (RD). The traces cover a wide range of complexity. The number of events in the traces ranges from 97 to 26612. The number of write events and read events ranges between 3 to 6213, 13 to 2163, respectively.

Race detection capability. The actual number of the races depends on the number of write and read events which in turn depend on the controller running the application. For example, on the same star topology, the LearningSwitch application running on Pox EEL generated 7 writes and 42 reads, but lead to 7 writes and 70 reads when running on Floodlight. Many detected races are harmless. If we report them to the developer, it will waste their time to fix these races, and may introduce real races. We used the commutativity and time filters proposed in [10] to filter these harmless races. The number of races after filtering is used to compare the race detection capability of SDN-predict and SDNRacer. Columns 7–10 present the metrics of races detected by SDN-predict: the total number of races (Race), races filtered by commutativity check (Comm.), races filtered by timimg (time), final races (Remain.). Columns 11–14 present the metrics of races detected by SDNRacer, their meanings are the same as columns 7–10.

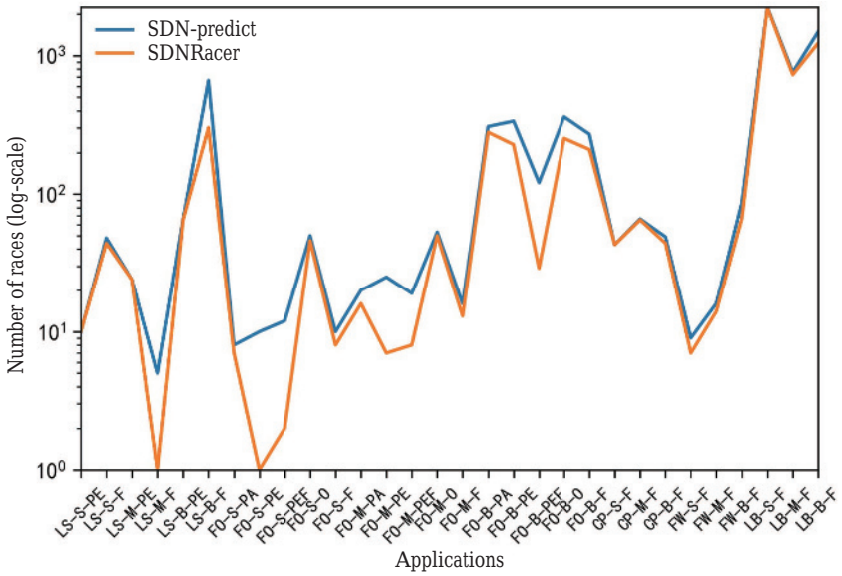

Figure 6 (Color online) Race detection capability of the two detectors. LS: LearingSwitch, FO: Forwarding, CP: CircuitPusher, FW: FireWall, LB: LoadBalancer, S: Star, M: Mesh, B: BinTree, PE: Pox EEL, F: Floodlight, PA: Pox Angler, PEF: Pox EEL Fx, O: ONOS.

As the results shown in Figure 6, for every application, our framework is able to detect more or at least the same number of races as SDNRacer. For the LearningSwitch application running by Pox EEL on the star topology and the CircuitPusher application running by Floodlight on star topology, SDN-predict detected the same number of races as SDNRacer. For the rest of the applications, SDN-predict detected more races than SDNRacer, ranging from 1 to 359. In total, SDN-predict detected 1173 race conditions that SDNRacer have not detected. For instance, for the LearningSwitch application running by Floodlight on BinaryTree topology, SDN-predict detected 661 races, while SDNRacer detected 302 races. This demonstrates that our framework achieves a higher race detection capability. The reason is that SDNRacer treats the conflicting events which do not exist paths between them (that is, no HB relations) as races, while SDN-predict is able to use the constraint solver to get all permutations of the events in the traces while satisfying the consistency requirements of trace, through reordering, the conflicting events with HB relations in the original trace τ can be adjacent in the τ-feasible traces, so they can be races detected by SDN-predict.

Next, we take a fragment of a trace in Floodlight_FireWall_StarTopology-steps200 to explain in detail why SDN-predict can detect 9 races, nevertheless SDNRacer can only detect 7 of them. The fragment of the trace is consisted of the following events: 127-133-136-141-139-146-150-152-155-161-164-170–168-176-174-181-185-188-190-196-200-202-205-211-214-220-218-226-224-231-235-238-240-246-250-252-255-261-265-267-273-277-280-282-285-291-295-298-300. In this trace, events 127, 155, 190, 205, 240, 255, 267, 285 and 300 are read events, events 168 and 218 are write events. SDN-predict get the MHB relations according to the rules defined in Table 2. The MHB relations in the trace include: $127 \prec 133$, $133 \prec 136$, $136 \prec 141$, $141 \prec 139$, $139 \prec 146$, $146 \prec 150$, $150 \prec 152$, $152 \prec 155$, $155 \prec 161$, $161 \prec 164$, $164 \prec 170$, $164 \prec 176$, $170 \prec 168$, $176 \prec 174$, $174 \prec 181$, $181 \prec 185$, $188 \prec 190$, $190 \prec 196$, $196 \prec 200$, $200 \prec 202$, $202 \prec 205$, $205 \prec 211$, $211 \prec 214$, $214 \prec 220$, $214 \prec 226$, $220 \prec 218$, $226 \prec 224$, $231 \prec 235$, $238 \prec 240$, $240 \prec 246$, $246 \prec 250$, $250 \prec 252$, $255 \prec 261$, $261 \prec 277$, $265 \prec 267$, $267 \prec 273$, $273 \prec 280$, $280 \prec 282$, $282 \prec 285$, $285 \prec 291$, $291 \prec 295$, $298 \prec 300$. SDNRacer can get the same MHB relations from the trace. After running SDNRacer and SDN-predict, SDNRacer detected the race conditions $(190, 168)$, $(240, 168)$, $(267, 168)$, $(127, 218)$, $(255, 218)$, $(285, 218)$, $(300, 218)$, SDN-predict can detect another two race conditions $(155, 168)$ and $(205, 218)$. The 7 race conditions can be detected by SDNRacer because the read-write pair in each race is unordered by MHB relation and other read-write pairs such as $(255, 168)$ are filtered by commutativity check. SDN-predict detected the 7 race conditions found by SDNRacer

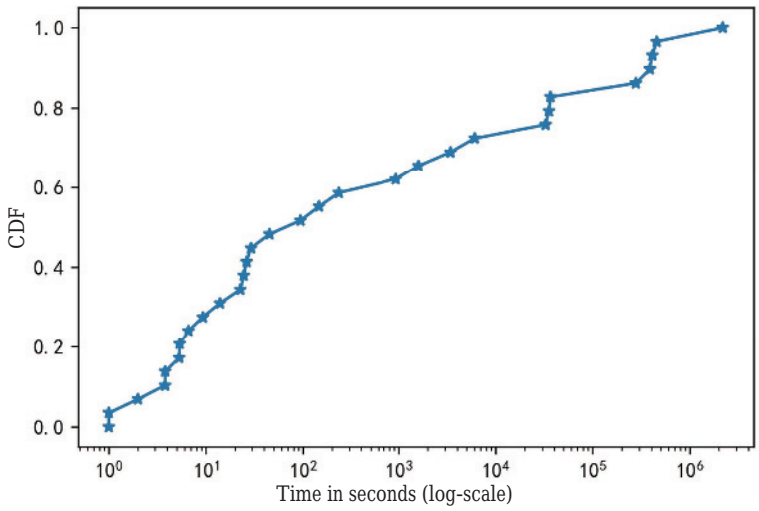

Figure 7 (Color online) Analysis time for traces from Table 3.

through reordering the events in the trace while keeping the consistency of the trace. For example, the trace can be reordered as ... 218-127 ... because there is no MHB relation between event 218 and event 127. SDN-predict can detect other races in the same reordering and filtering procedure as detecting $(127, 218)$. For race conditions $(155, 168)$ and $(205, 218)$, these two event pairs can be ordered by MHB relations, so SDNRacer cannot detect them. However, events 161 and 211 are asynchronous events, SDN-predict can reorder them in the trace as long as the result trace is feasible. The feasible traces can be: ... 152-161-164-170-168-155 ... and ... 202-211-214-220-218-205 .... SDN-predict can detect these two races by constraint solving.

Also, our framework is sound and maximal which is proved in last section. In the stage of constraints construction, the constraints are constructed based on the HB graph which specifies the HB relation between events. Many races reported initially by our framework are harmless. We use the commutativity and time filters proposed in [10] to filter these harmless races. Commutativity filtering can filter the harmless race if the two events are for non-overlapping flow entries of the flow table and the network state is not affected if we change their order. In theory, SDN switches can take an unbounded amount of time to perform an event. Time filtering can filter the harmless race if the distance between the two events in the original trace exceeds the time window (for instance, if a read and a write event are separated by 2 s, then they are unlikely interfere). The filter efficiency is evaluated in [10]. The filters can filter more than 90% harmless races. Furthermore, the 1173 races detected by our framework but have not detected by SDNRacer are confirmed by checking the race graphs. Note that any dynamic race detection technique (including ours) is sensitive to the analysed trace. The results reported for different traces are incomparable. Therefore, it is possible for our framework to miss some races reported in other studies for the same application running by the same controller on the same topology, because the traces in our experiments may be different from those used in other's work.

Scalability. We analyse the symbolic complexity of our predictive analysis framework to illustrate its scalability and limitation. The predictive analysis for race detection is divided into two phases: constraint encoding and constraint solving. Assume the number of events in the trace is n, where the number of read events is k and the number of write events is l. The constraint coding phase includes HB graph building and constraint constructing. The time complexity of them are both $O(n^2)$. The time complexity of constraint solving is $O((k \times l + l^2) \times n \times m)$, where m is the number of constraints. Thus, the total time complexity of our predictive analysis framework is $O(2n^2 + (k \times l + l^2) \times n \times m)$. The performance of our framework largely depends on trace size (the number of events in trace), the number of conflicting pairs (read and write events) in the traces, the number of constraints and the speed of the constraint solver. Our framework shows good scalabitity in most cases. As shown in Figure 7, for two-thirds of the cases SDN-predict can analyze the traces ranging from 1 s to a few minutes. In the remaining 1/3

cases, SDN-predict needs 0.95 to 607 h to finish the trace analysis. Particularly LoadBalancer running by Floodlight on BinaryTree topology takes the longest time to finish. The reason is that these cases have a lot of conflicting pairs, making the generated constraints much more complex, and the traces contain large number of events, leading to a number of reorderings. So our framework can perform well if the trace size and the number of the conflicting pairs are not very large.

For all the applications, SDNRacer fininshed in few seconds or minutes which is faster than our framework. This is because SDNRacer traverses the HB graph to check whether there exists a path for each conflicting pair in the trace τ. However, our framework relys on SMT solving and explores all the τ-feasible traces for each conflicting pair.

6 Related studies

SDN race detection. There is a lot of race detection work for SDN in the literature. SDNRacer [10] is a representative approach in this direction and inspires our work. Both techniques aim at detecting races between the SDN controller and the SDN swithes. SDNRacer uses the HB model to detect race condition in trace τ, while our framework uses contraint solver to detect the races in all τ-feasible traces. So our framework can find races cannot be detected by SDNRacer. And the HB-based dynamic analysis technique may have false positives, while the races reported by our framework are all confirmed by checking the race graphs. CONGUARD [4] is another HB model based race detector for SDN. The objective of this framework is to detect the race conditions in the SDN controllers. However SDN-predict is used to predictive analysis the race conditions between the controller and the switches, that is races on flow tables. Attendre [21] identifies the race conditions and mitigates the ill effects of the race conditions on verification of the SDN controller applications. This tool uses model checking to detect several types of violation not including race conditions. Attendre and other verification tools [22, 23] for SDN cannot tell the sequence of events leading to violations. Yet, SDN-predict can give the sequence of events corresponding to the races. BigBug [24] is an approach for identifying the most representative concurrency violations for which reported by the SDN concurrency analyzers. They cluster reported violations and report the most representative violation for each cluster.

Static analysis. Static analysis is a kind of important technique for race detection in concurrenct programs. Static analysis analyses the whole program without executing source code. It usually checks whether the memory locations accessed concurrently are alias, local or protected by the same lock set using the alias analysis, thread escape analysis and lockset analysis. Static analysis techniques for finding concurrency bugs either sacrifice precision for performance, leading to many false positives. Another method proposed by [25] employs a combination of static analysis approaches to reduce the pairs of memory accesses potentiallly involved in a race. Ref. [26] combines the static analysis with bug pattern to produce a low false positive rate. Several static analysis tools have been developed for detecting concurrency bugs[6)7)8)].

Dynamic analysis. Dynamic analysis for race detection [27–30] needs to instrument the source code and collect the execution information. It uses the lockset or the HB relations and its variations [31, 32] or both to detect the races in the execution trace. Dynamic analysis is precise but incomplete, for it cannot cover all possible traces, and high overhead caused by instrumentation. Ref. [33] is based on binary rewriting techniques to monitor every shared memory reference and to verify that consistent locking behavior was observed. FastTrace [34] uses an adaptive representation for the happens-before relation to detect the data race. RaceFuzzer [35] computes a set of pairs of statements that could potentially race during a concurrent execution, then uses each element from the set to control the random scheduling of the threads. Hybrid dynamic race detection [36] combines the lockset-based and HB-based approach. It uses lockset-based method to get the initial race set, then uses HB-based race detection to detect races

6) FindBugs. http://findbugs.sourceforge.net/.
7) Jlint. http://jlint.sourceforge.net/.
8) jChord. http://code.google.com/p/jchord/.

from this set to reduce the number of false positives by lockset analysis. There also several dynamic analysis tools for race detection have been developed[9)10)11)12)].

Predictive analysis. Our framework belongs to the category of predictive analysis approach, which generates valid trace reordering under certain constraints to find bugs undetected in original trace. Existing predictive analysis are originally used to detect the data race and atomicity violation for concurrency program such as Java and C [6–9]. They encode the constraints according to the program structures, and focus on detecting the concurrency issues on memory locations. As far as we know, it is the first time to apply predictive analysis to SDN race detection. Our framework encodes the constraints based on the events contained in the network devices, and aims to find the races on flow tables.

7 Conclusion

In this paper, we present a predictive analysis framework SDN-predict for race detection in SDNs. We formulate the race detection as a constraint solving problem over a set of constraints. We encode the constraints based on the order between the network events and their dependencies, and use a SMT solver Z3 to find all real races in all valid reordering traces. We have conducted extensive experiments with our framework and compared its race detection capability with SDNRacer. The results demonstrate that our framework has higher race detection capability and is scalable in detecting races in large networks.

Acknowledgements This work was partially supported by National Basic Research Program of China (973 Program) (Grant No. 2014CB340702), National Natural Science Foundation of China (Grant Nos. 91418202, 61472178, 91318301), and National Science Foundation for Young Scientists of China (Grant No. 61702256).

References

1 Open Networking Foundation. OpenFlow Switch Specification. version 1.3.3. 2013. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.3.pdf
2 Deng D D, Jin G L, de Kruijf M, et al. Fixing, preventing, and recovering from concurrency bugs. Sci China Inf Sci, 2015, 58: 052105
3 Wu Z D, Lu K, Wang X P. Surveying concurrency bug detectors based on types of detected bugs. Sci China Inf Sci, 2017, 60: 031101
4 Xu L, Huang J, Hong S M, et al. Attacking the brain: races in the SDN control plane. In: Proceedings of the 26th USENIX Security Symposium, Vancouver, 2017. 451–468
5 Cai Y, Cao L W. Effective and precise dynamic detection of hidden races for Java programs. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Bergamo, 2015. 450–461
6 Huang J, Meredith P O, Rosu G. Maximal sound predictive race detection with control flow abstraction. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, 2014. 337–348
7 Huang J, Zhou J G, Zhang C. Scaling predictive analysis of concurrent programs by removing trace redundancy. ACM Trans Softw Eng Method, 2013, 22: 1–21
8 Liu P, Tripp O, Zhang X Y. IPA: improving predictive analysis with pointer analysis. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, 2016. 59–69
9 Wang C, Kundu S, Limaye R, et al. Symbolic predictive analysis for concurrent programs. Form Asp Comput, 2011, 23: 781–805
10 El-Hassany A, Miserez J, Bielik P, et al. SDNRacer: concurrency analysis for software-defined networks. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, Santa Barbara, 2016. 402–415
11 Zhang Z Y, Chen Z Y, Gao R Z, et al. An empirical study on constraint optimization techniques for test generation. Sci China Inf Sci, 2017, 60: 012105
12 Big Switch Networks, Inc. Floodlight learning switch. 2013. https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/learningswitch
13 McCauley J. POX EEL L2 learning switch. 2015. https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_learning.py

9) ConTest. http://www.haifa.ibm.com/projects/verification/contest/.
10) CalFuzzer. http://srl.cs.berkeley.edu/~ksen/calfuzzer/.
11) CHESS. http://research.microsoft.com/en-us/projects/chess/.
12) Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf.

14 Big Switch Networks, Inc. Floodlight forwarding application. 2013. https://github.com/floodlight/floodlight/blob/v0.91/src/main/java/net/floodlightcontroller/forwarding/Forwarding.java
15 McCauley J. POX angler forwarding application. 2012. https://github.com/noxrepo/pox/blob/angler/pox/forwarding/l2_multi.py
16 McCauley J. POX EEL forwarding application. 2015. https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_multi.py
17 Open Networking Laboratory. ONOS: forwarding application. 2015. https://github.com/opennetworkinglab/onos/tree/onos-1.2/apps/fwd
18 Big Switch Networks, Inc. Floodlight circuit pusher application. 2013. https://github.com/floodlight/floodlight/tree/v0.91/apps/circuitpusher
19 Big Switch Networks, Inc. Floodlight firewall. 2013. https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/firewall
20 Big Switch Networks, Inc. Floodlight load-balancer application. 2013. https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/loadbalancer
21 Sun X S, Agarwal A, Ng T S E. Controlling race conditions in OpenFlow to accelerate application verification and packet forwarding. IEEE Trans Netw Serv Manage, 2015, 12: 263–277
22 Majumdar R, Tetali S D, Wang Z. Kuai: a model checker for software-defined networks. In: Proceedings of the 14th International Conference on Formal Methods in Computer-Aided Design, Portland, 2014. 163–170
23 Khurshid A, Zhou W, Caesar M, et al. Veriflow: verifying network-wide invariants in real time. SIGCOMM Comput Commun Rev, 2012, 42: 467–472
24 May R, EI-Hassany A, Vanbever L, et al. BigBug: practical concurrency analysis for SDN. In: Proceedings of the Symposium on SDN Research, Santa Clara, 2017. 88–94
25 Naik M, Aiken A, Whaley J. Effective static race detection for Java. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, 2006. 308–319
26 Luo Z D, Hillis L, Das R, et al. Effective static analysis to find concurrency bugs in Java. In: Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation, Timisoara, 2010. 135–144
27 Pozniansky E, Schuster A. Efficient on-the-fly data race detection in multithreaded C++ programs. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, 2003. 179–190
28 Serebryany K, Iskhodzhanov T. ThreadSanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, New York, 2009. 62–71
29 Xie X W, Xue J L. Acculock: accurate and efficient detection of data races. In: Proceedings of the 9th International Symposiumon Code Generation and Optimization, Nanjing, 2011. 201–212
30 Yu Y, Rodeheffer T, Chen W. RaceTrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings of ACM Symposium on Operating Systems Principles, Brighton, 2005. 221–234
31 Yannis S, Jacob M E, Caitlin S, et al. Sound predictive race detection in polynomial time. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Philadelphia, 2012. 387–399
32 Dileep K, Umang M, Mahesh V. Dynamic race prediction in linear time. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, 2017. 157–170
33 Savage S, Burrows M, Nelson G, et al. Eraser: a dynamic data race detector for multithreaded programs. ACM Trans Comput Syst, 1997, 15: 391–411
34 Flanagan C, Freund S N. FastTrack: efficient and precise dynamic race detection. In: Proceedings of the 30th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, 2009. 121–133
35 Sen K. Race directed random testing of concurrent programs. In: Proceedings of the 29th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, 2008. 11–21
36 Callahan R, Choi J D. Hybrid dynamic data race detection. In: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, California, 2003. 167–178