



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Institut für
Technische Informatik und
Kommunikationsnetze

Roman May

Practical Concurrency Analysis for SDN

Master's Thesis
June to December 2016

Supervisor/Tutor: Prof. Dr. Laurent Vanbever
Co-Tutor: Ahmed El-Hassany

Abstract

By operating in highly asynchronous environments, SDN controllers often suffer from bugs caused by concurrency violations. Unfortunately, state-of-the-art concurrency analyzers for SDNs often report thousands of possible violations, limiting their effectiveness in practice.

This work presents **BigBug**, an approach for automatically identifying the most representative concurrency violations: those that capture the cause of the violation. The two key insights behind **BigBug** are that: *(i)* many violations share the same root cause; and *(ii)* violations with the same root cause share common characteristics. **BigBug** leverages these observations to cluster reported violations according to the similarity of events in them as well as common SDN-specific features. **BigBug** then reports the most representative violation for each cluster using a ranking function.

We implemented **BigBug** and showed its practical effectiveness. In more than 2000 experiments involving different controllers and applications, **BigBug** systematically produced 6 clusters or less, corresponding to a median decrease of 95% over state-of-the-art analyzers. The number of violations reported by **BigBug** also closely matched that of actual bugs, indicating that **BigBug** is effective at identifying root causes of SDN races.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Challenges and contributions	3
1.3. Related work	3
1.4. Thesis structure	4
2. Overview	5
2.1. Motivating example	5
2.2. BigBug	7
3. Pre-processing	9
3.1. Trimming SDNRacer HB-graph	9
3.2. Extracting per-violation graphs	10
4. Clustering	12
4.1. Cluster initialization	12
4.2. Identifying related violations through SDN-specific features	14
4.3. Distance calculation	16
4.4. Clustering algorithm	17
4.5. Discarded approaches	19
4.5.1. Discarded clustering algorithms	20
4.5.2. Discarded distance metrics	21
5. Ranking	24
6. Evaluation	27
6.1. Experimental setup	27
6.2. Usability	29
6.3. Use case	32
6.4. Influence of trace length on BigBug	34
6.5. Performance	35
7. Outlook	38
8. Conclusion	40
Bibliography	41
Appendix A. Complete evaluation table	44
Appendix B. Declaration of originality	46

List of Figures

1.	Number of violations reported by SDNRacer compared to BigBug	2
2.	Example concurrency violation in Floodlight Load Balancer module	5
3.	Example HB-graph	6
4.	Working pipeline for BigBugs concurrency analysis	8
5.	Example trimming HB-graph	10
6.	Example extraction per-violation graph	11
7a.	Example isomorphic cluster initialization (isomorphic)	13
7b.	Example isomorphic cluster initialization (not isomorphic)	13
8.	Example weakly connected components	22
9.	Example ranking function	26
10.	CDF of the % of violations reduced by BigBug	31
11.	Usecase example graph	32
12.	CDF of the execution time of BigBug	35

List of Tables

1.	Example cluster features	15
2.	Example distance matrix	17
3.	Example updated distance matrix	19
4.	Example ranking features	24
5.	Feature weights for evaluation	28
6.	Evaluation of 200 step traces	30
7.	Comparison of original and fixed Floodlight Load Balancer module	33
8.	Influence of the trace length on the number of final clusters.	34
9.	Execution times of SDNRacer and BigBug on 200 steps traces	36

Listings

1.	k-medoids algorithm	20
2.	DBScan algorithm	21

1. Introduction

1.1. Motivation

Software Defined Networking (SDN) [1] is a recent approach to improve today’s communication networks. The key idea is to separate the control plane from the data plane, and orchestrate the network from a single point of control, called *SDN controller*. Further, SDN provides different abstractions for forwarding, state-distribution and specifications to the network administrator. The data plane consists of *SDN switches* that communicate with the controller via OpenFlow [2]. When a packet arrives at a switch, the switch forwards it based on the entries in its forwarding table. If there is no matching rule in the table, the switch dispatches the packet to the SDN controller, which then adds, modifies, and removes entries in the switches flow tables. The controller orchestrates the traffic in the network either proactively, e.g., network initialization, or reactively, e.g., packets sent from switches or flow expiry messages. Further, it provides a well-defined API for networking applications.

Even though SDN is a promising approach, it can also be a source of new difficulties: SDN controllers operate in highly asynchronous environments where events such as packets arriving at a switch or expiring flows can be dispatched to the controller at any time, non-deterministically. Programming highly asynchronous programs is known to be hard. In particular, interfering accesses to shared variables (e.g., a switch forwarding table) can often lead to unwanted behaviors and bugs. A classic example is an SDN controller which modifies the content of a forwarding table according to the packets it sees and its internal state. Depending on the order in which writes (FLOW_MOD) and reads (PACKET_IN) occur, the forwarding state of the switch can differ dramatically.

Recently, SDNRacer [3, 4], a dynamic concurrency analyzer, showed that it is possible to identify concurrency violations in existing SDN controllers. At its core, SDNRacer is based on a Happens-Before (HB) model, a specification of how different OpenFlow events are ordered. Given a trace of OpenFlow events and the HB model, SDNRacer builds a dependency graph (HB-graph) which it uses to detect concurrency violations.

SDNRacer runs on traces generated by STS [5], which simulates a SDN network, uses real SDN controller to control the networks and records all events happening in the network. In the whole process, the controller is treated as a black box, making it compatible with any current or future controller.

Problem Statement While a precise concurrency analyzer (such as SDNRacer) is a useful first step, if used only by itself, it (or any other state-of-the-art concurrency analyzer) is not yet a complete practical solution to the problem of identifying the key concurrency violations in an SDN setting. The fundamental reason is that there are too many (thousands) concurrency violations even for short traces of few seconds. These violations are *not* false positives—these are real violations that can actually occur—

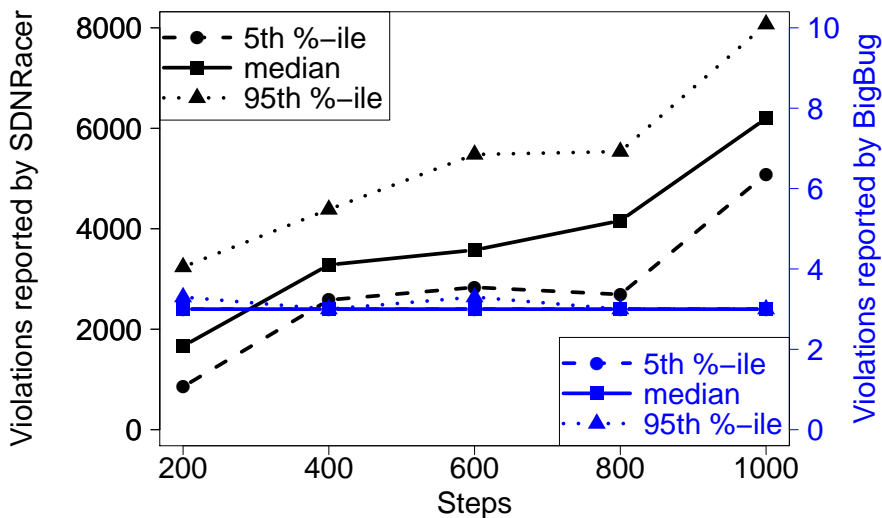


Figure 1: Even when considering one SDN application (Floodlight Load Balancer) running in a star topology with 4 hosts, the number of concurrency violations reported by SDNRacer is huge and of little practical use. BigBug reduces it to 3 representative violations which, when fixed, make more than 99.6% of the violations disappear.

and thus a HB sound concurrency analyzer (like SDNRacer) will report them. As an illustration, Figure 1 depicts the number of violations reported by SDNRacer as a function of the trace length collected on a Floodlight [6] controller running the default Load Balancer application [7]. A trace of 200 steps corresponds to ~ 30 seconds. We can see that a 1000 steps trace (~ 2.5 minutes) generates no less than *6,000 distinct* concurrency violations! Clearly, trying to sift through such a high number of (actual) violations is practically infeasible. Further, we expect the number of violations to be much higher in practice as SDN controllers tend to run more than one application.

Intuitively, however, the number of causes that trigger these violations should (hopefully) be orders of magnitude less, meaning that many violations originate from the same cause (i.e., the same bug). Indeed, in Figure 1, the controller only contains two distinct bugs. While different violations resulting from the same cause will differ in some ways (at the very least, because different packets trigger them), one can expect that they share a common structure. Yet, even (sometimes, highly) similar, violations will also share distinctions making it hard for a human to distinguish them. While a lot of these violations are a result of the same bug, the output is not ordered or filtered in any way and it is the duty of the developer to painfully go through each of them and reason about their root causes.

This work The key idea of our work is an approach and a framework which can automatically process thousands of SDN concurrency violations and identify the most representative ones. BigBug takes as input a set of violations reported by a concurrency analyzer (e.g., SDNRacer or similar) and clusters these reports into equivalence classes. BigBug then selects the most representative violation in each class and presents it to the

SDN developer. The developer can then focus on understanding the root cause of that violation, knowing that thousands of others share the same characteristics. The blue part of Figure 1 illustrates the benefits of **BigBug**: while the number of violations reported by **SDNRacer** grows linearly, **BigBug** automatically reduces it down to 3 equivalence classes. More importantly, 99.66% of the violations disappear on average after fixing the root cause behind each of them.

1.2. Challenges and contributions

Identifying the most representative violations among 1,000s is challenging for at least two reasons. First, to define a cluster, we need to define a notion of *distance* between two distinct violations. Here, **BigBug** uses an isomorphism-based approach to initialize the clustering process using “look-alike” violations. Yet, as many similar violations are not isomorphic, **BigBug** leverages feature-based clustering derived from domain-specific knowledge of SDN networks. Second, after a cluster is determined, **BigBug** selects the most representative candidates for each cluster so as to maximize usefulness for the developer.

We implemented **BigBug** and evaluated its effectiveness. We show that it successfully reduces **SDNRacer** output by orders of magnitude on average.

Overall, we believe that the techniques behind **BigBug** may be applicable in many other contexts where many graphs have to be sorted out and the most representatives ones selected. One example among others is HTTP request filtering so as to identify outliers among millions of Web requests [8].

Our main contributions are:

- A set of domain-specific features to measure the similarities between reported concurrency violations (sections 4.1 and 4.2).
- A novel technique to cluster related concurrency violations using the set of domain-specific features (chapter 4).
- A set of ranking techniques which allows **BigBug** to select a representative candidate of each cluster (chapter 5).
- A complete implementation of **BigBug** along with a comprehensive evaluation where we show its practical relevance: **BigBug** systematically reported 6 violations or less in more than 2000 experiments. In a case study, we also show that solving the bug behind the reported violations caused 99.66% of them to disappear (chapter 6).

1.3. Related work

As mentioned before, **BigBug** goes beyond current SDN specific concurrency analyzers like **SDNRacer**, and clusters related concurrency violations. To the best of our knowledge, this is a novelty for the SDN domain.

However, filtering or clustering of concurrency violation reports has been applied for event-driven concurrency analyzers in other domains before, as for example [9] shows. [10]

uses a HB model to find concurrency violations in Android applications and groups them based on event sources (e.g. threads, input, etc.), while [11] uses a lightweight HB representation to find concurrency violations in multithreaded programs. [12], [13] and [14] cluster reported alarms of static code analyzers by finding dependencies and correlations between alarms. Further, [15] finds relations between different reported warnings in a cluster-code analyzer, so as to reduce the redundancy.

Similar to those works, our work clusters reported violations to reduce the developers effort, but mainly differs as our clustering method is based on fine-grained semantic HB information rather than coarse-grained indicators (e.g., whether an operation in a violation is in the framework [10]). Also, the use of domain-specific features of violation graphs to measure similarities between them is unique among this works.

Moreover, our work does not rely on static analysis and actually considers the controller code as a black box, as opposed to static code analyzers. Thanks to this, our clustering approach based on HB information is general and can thus benefit existing analyzers such as [10].

BigBug also goes beyond reducing the number of false positives produced by traditional concurrency analyzers by automatically reasoning about the common causes underlying the violations using domain-specific knowledge. Although there are tools available that also classify concurrency violations, like [11] and [16], they use known classes to group violations, while **BigBug** dynamically clustering concurrency violations based on domain-specific features, which allows it to classify the violations based on, potentially unknown, root causes.

1.4. Thesis structure

The remainder of this report is structured as follows: an overview of **BigBug** is provided in chapter 2. We explain the processing pipeline of **BigBug**, the pre-processing, the clustering and the ranking, in chapters 3, 4, and 5, in detail. The evaluation of **BigBug** is covered in chapter 6, and suggestions for future work are provided in chapter 7. We complete this report with a conclusion in chapter 8.

2. Overview

In this section, we provide a high-level overview of **BigBug**. We start with a motivating example (section 2.1), illustrating how several concurrency violations can result from the same bug in the SDN controller. We then highlight how **BigBug** (section 2.2) clusters a large amount of concurrency violations into few representative clusters.

2.1. Motivating example

We consider a Floodlight controller running the default Load Balancer application which redirects Web requests to two replicas in a round-robin fashion (Figure 2). We assume that an external host, say Host#1 (H1), sends a Web request, we call this a Host Send event, which hits S1 ①. As it is a new request there is no matching forwarding entry in S1's flow table ②, and thus S1 directs it to the controller using an OpenFlow `PACKET_IN` message ③. The controller selects the best replica, say Replica#1 (R1), and sends three OpenFlow messages to S1. The first one is a `FLOW_MOD` to install a new forwarding entry to forward packets coming from H1 to R1 ⑤. The second message is also a `FLOW_MOD` that forwards packets from R1 to H1 ⑥. The third message is a `PACKET_OUT` that carries the original packet sent to the controller ④. S1 relies on its flow table to forward that packet, since the controller assumes that there are entries already installed to match on this packet.

According to the specification [2], S1 can process these three events in any order unless separated by barrier messages. A possible execution is therefore S1 processing the `PACKET_OUT` before the two `FLOW_MOD` messages. In this case, S1 does not have any flow entry matching on the packet, and it sends another `PACKET_IN` message to the controller as a result ⑦. This “ping-pong” effect lasts until S1 installs the flow entry

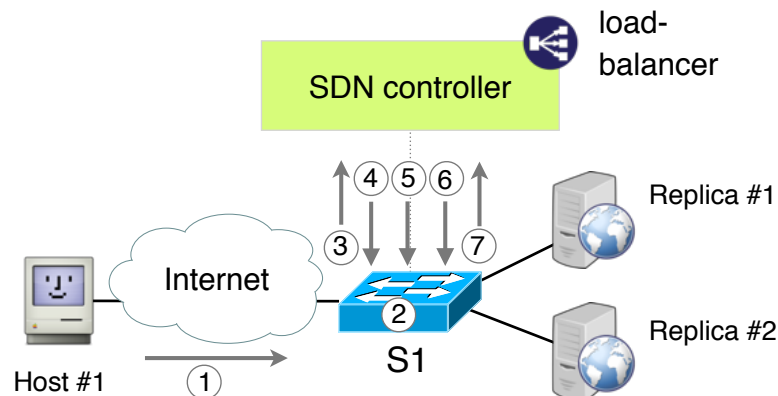


Figure 2: One of the many sequence of events creating concurrency violations in the default Floodlight Load Balancer application.

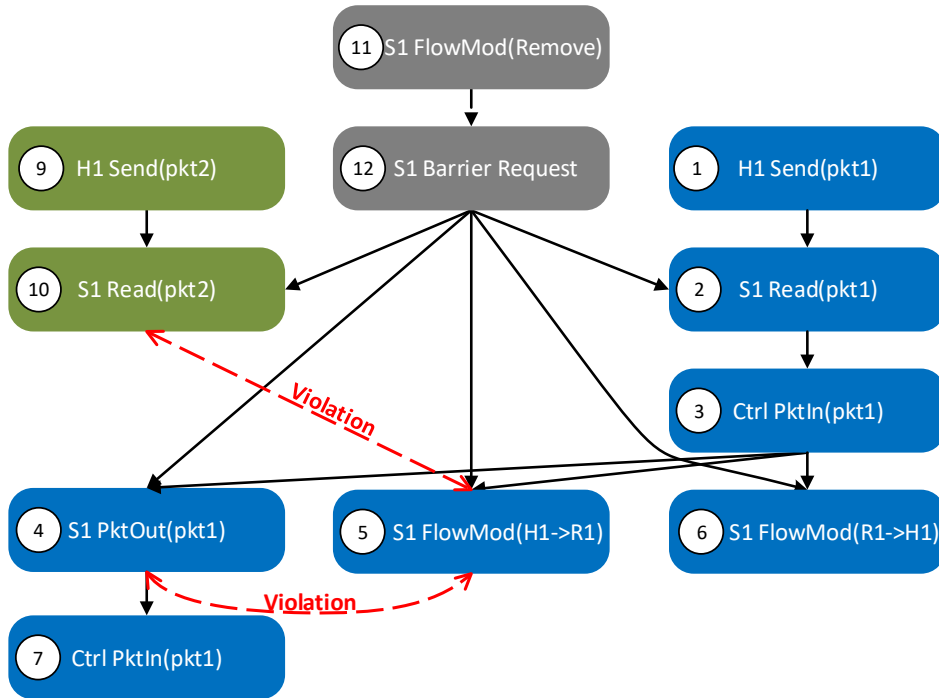


Figure 3: A simplified example of a possible HB-graph created by SDNRacer for a trace of the Floodlight Load Balancer module.

directing packets from H1 to R1. In addition to the obvious inefficiencies, this behavior can also create serious forwarding issues such as non-deterministic load-balancing between the two replicas (effectively killing the connection) or forwarding loops [3, 4].

SDNRacer detects and reports all these violations—every single one of them—which can amount to thousands even in minutes-long traces. Analyzing and troubleshooting all these violations is tedious for at least three reasons. *First*, violations originating from the same bug might differ (either subtly or vastly), which makes them hard to classify manually. In our previous example, violations would differ in the number of “bounces” observed between the controller and the switch. Worse, multiple switches can also be involved leading to a combinatorial explosion in the number of distinct violations. *Second*, the number of violations induced by each bug can vary significantly (chapter 6) and does not necessarily correlate with the importance of the problem. For instance, $\sim 90\%$ of the violations reported when running the above controller originate from the relatively benign bug explained in the motivating example (section 2.1). *Third*, a developer has no information on the number of bugs that are causing the violations. A naive approach to randomly fix one violation at a time and then re-run the analysis can therefore take a long time to converge and be sub-optimal (especially, if done in a greedy way).

As explained in the previous chapter, SDNRacer first builds an HB-graph. Figure 3 depicts a simplified section of the HB-graph for the previous example. Normally, a HB-graph consists of thousands of events, even for short traces. The rectangles illustrate single events, while the directed edges provide information about the ordering of the

events. Note, that the dotted red violation connectors are not considered edges, as they do not indicate an ordering of the events. They only serve to point out the events that are part of the concurrency violation. Further, the event IDs in the graphs $\#$ are only used to identify different events, and do not indicate an ordering of events.

The grey events ⑪⑫ are part of the network initialization. Most controllers first remove all flow rules from the flow tables of all switches and send a synchronization barrier to bring the whole network to a known state.

The blue events ①-⑦ illustrate the events described in the Load Balancer example above. SDNRacer first marks all combinations of the two FLOW_MOD⑤⑥ and the PACKET_OUT④ as potential harmful concurrency violations. Since the FLOW_MOD that adds the return path from R1 to H1 ⑥ has a different match than the other two messages, SDNRacer will only report a concurrency violation between the PACKET_OUT④ and the FLOW_MOD⑤ that adds a rule to forward packets from H1 to R1.

The green events ⑨⑩ show what happens if the same host sends another request shortly after the first one. There is now way to tell if the resulting flow table read at S1 ⑩ is processed before or after the FLOW_MOD⑤. Therefore, SDNRacer also reports a concurrency violation between these two events.

We will use the above example, in the following chapters to illustrate each step of the processing pipeline of BigBug. All paragraphs that begin with **Example** refer to this example of the Load Balancer application.

2.2. BigBug

To aid the debugging process, BigBug aims to present the developer with only the representative violations which, ideally, corresponds to the actual bugs. This allows SDN developers to focus on addressing the most serious cases. BigBug reduces the number of concurrency violations according to a three step process (Fig. 4).

Step 1: Pre-processing Out of a given execution trace, a concurrency analyzer (e.g., SDNRacer) will typically build a directed graph according to a Happens-Before (HB) relationship (where event a is connected to b if a happens before b). The analyzer will then report a concurrency violation for any two events which are unordered in the graph (i.e., are disconnected), both events access the same location, and one is a FLOW_MOD(write event).

As BigBug needs to compare violations together, a pre-processing step first produces one sub-graph per violation given the HB-graph. This sub-graph only contains the events that led to the violation.

Step 2: Clustering Given a set of per-violation graphs, BigBug clusters these graphs into a number of (ideally, representative of the bugs) classes. BigBug initializes the clustering process by grouping all isomorphic per-violation graphs. The intuition is that because these graphs share the same sequence and structure of events, they are more likely to exert the same code path (and therefore the same bug) inside the controller.

While isomorphic-based clustering is efficient at identifying “look-alike” violations, different violations from the same bug can take different shapes (as we illustrated in section 2.1). For instance, in the bug we described in the Load Balancer application, the packet

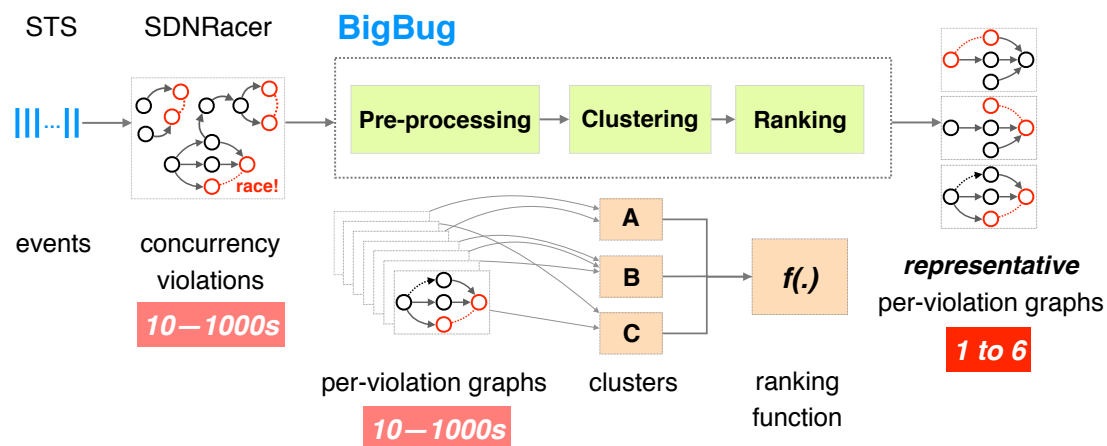


Figure 4: A practical working pipeline for SDN concurrency analysis. Out of potentially thousands of violations reported by SDNRacer, BigBug reduces them to a handful of representative ones which closely map to actual controller bugs.

might bounce between the controller and the switch more than one time before the forwarding entry is committed to the flow table. Just using isomorphic clustering will group only the races with the same number of bounces between the switch and the controller.

Therefore, in the second phase, to reduce the number of clusters, BigBug applies a clustering strategy based on whether two per-violation graphs are similar to each other. BigBug defines this similarity based on distance defined over a set of domain-specific features. If two per-violation graphs exhibit the same features, they are considered similar to each other and are clustered together. BigBug uses several features for the distance computation, for instance, two violations are closer to each other if both have a packet bouncing between the controller and the switch (as described in our example).

Step 3: Ranking Since the number of clusters reported by BigBug is very low (6 or less in all our experiments), each of the clusters contains many violations, sometimes in the order of 1000s. In the final step of the pipeline, BigBug uses a ranking function to select “the most interesting” violation representative of the entire cluster. This is done by identifying the most commonly occurring features in each cluster. We then select the per-violation graphs that exhibit the most features and select the smallest of these, thus, showing the simplest representative graph.

In the next chapters, we explain each step of the processing pipeline of BigBug, the pre-processing (chapter 3), the clustering (chapter 4) and the ranking (chapter 5), in detail.

3. Pre-processing

BigBug starts by pre-processing the output of SDNRacer: the directed graph induced by the HB relationship (HB-graph) and a list of violations, to produce one graph per violation with only the events that led to it.

In section 3.1, we show how BigBug reduces the size of the HB-graph. Then, in section 3.2, we present how BigBug extracts sub-graphs to help analyzing each concurrency violation individually in later stages.

3.1. Trimming SDNRacer HB-graph

While the number of events in each trace is large, not all of these events pertain to concurrency violations, some of them are irrelevant. Such events are filtered by BigBug, so as to reduce the computational complexity of the following stages, especially the clustering stage.

BigBug removes three categories of events from the HB-graph. *First*, it removes all the events that occurred during the network initialization phase and did not cause any concurrency violations such as the handshake messages between each switch and the controller. *Second*, it removes all the events that do not lead to a concurrency violation. *Third*, it removes redundant HB edges in the HB-graph.

Network initialization The first task SDN controllers accomplish in a network is to bring all switches in the network to a known state. Normally, the controller first sends a FLOW_MOD messages to each switch that removes all flow rules from the forwarding table, followed by a synchronization barrier. With this, the controller resets each switches initial state and makes sure, that there are only rules in all forwarding tables that it knows about. As depicted in Figure 3, the synchronization barrier that follows the FLOW_MOD messages connects multiple events, and therefore makes operations on the graph more computational expensive. BigBug removes these events given that they do not take part in any concurrency violation.

Irrelevant events As we are only interested in the events that lead to a concurrency violation, we remove all events that do not lead to, nor are part of, a violation. Specifically, we recursively remove all leaf nodes, i.e. nodes with only incoming and no outgoing edges, in the graph if they do not take part in a concurrency violation until all leaf nodes of the HB-graph are violation events.

Redundant edges SDNRacer uses several rules to connect events with HB-edges, e.g. based on packet or message IDs. Therefore, the resulting HB-graph often has more than one path between two ordered events. As one path is completely sufficient to order any two events, the trimming removes redundant edges from the graph.

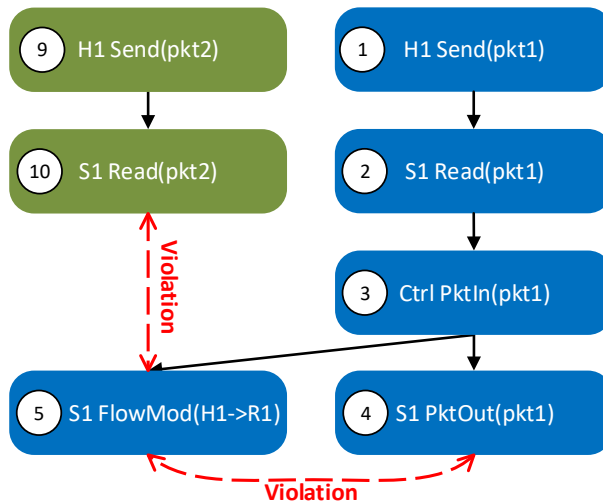


Figure 5: HB-graph from Figure 3 after the trimming step of the pre-processing stage.

Example Figure 5 shows the HB-graph in Figure 3 after the trimming. The trimming removes the two grey events ① and ⑫, as they are part of the network initialization and are not part of a violation. Further, events ⑥ and ⑦ neither take part in nor lead to a concurrency violation, so they are removed as well. In a not simplified version of the HB-graph in Figure 3, there would also be an additional edge between event ② and ④ as they share the same packet ID. The last step of the trimming would remove this edge, as the path ② → ③ → ④ already indicates that ④ happens after ②. However, as there are no redundant edges in our example HB-graph, this step does not remove any edges from the graph.

3.2. Extracting per-violation graphs

Even after removing irrelevant events, the resulting HB-graph is still massive containing many events and concurrency violations. As we are interested in how individual concurrency violations compare with each other, **BigBug** isolates each one of them into a separate graph such that each graph contains a single concurrency violation with all the events that led to it. **BigBug** builds the violation graphs by performing an upward traversal of the HB-graph starting from the two events involved in each violation until it reaches one of the entry points (,e.g. host send or proactive update) present in the trace. Note, that any single event might cause more than one violation, in this case, the event appears in multiple violation graphs.

Example Figure 6 illustrates the two per-violation graphs that **BigBug** extracts from the trimmed HB-graph in Figure 5. As mentioned before, there are events that contribute to both violations, and therefore, they are present in both violation graphs (i.e. ①-③

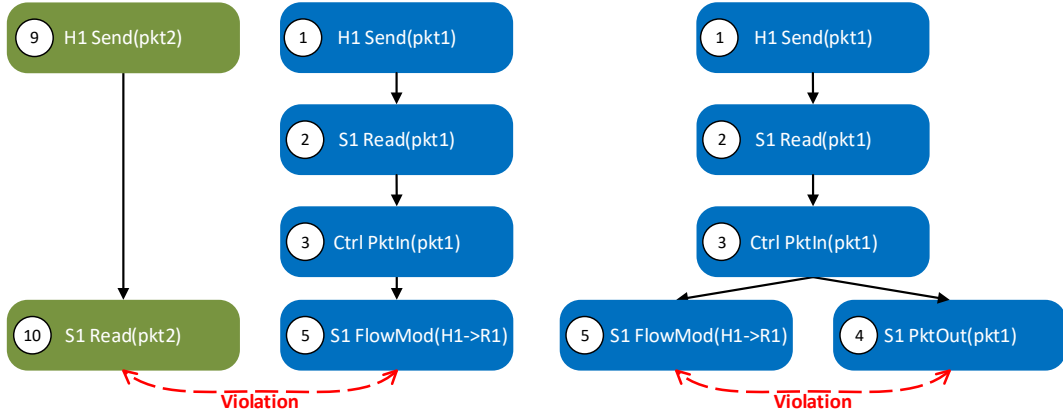


Figure 6: Extracted per-violation graphs from the trimmed HB-graph in Figure 5. Each graph contains exactly one concurrency violation and all events that led to it.

and ⑤).

The graph on the right side of the figure provides enough information for a developer to figure out that there is a missing synchronization barrier between the `FLOW_MOD` ⑤ and the `PACKET_OUT` ④ messages.

The graph on the left side on the other hand is an example of an *unfixable* violation. As the controller has no control on when a host in a network sends a packet, it is always possible that a packet from a host arrives at the switch at the same time as it receives a `FLOW_MOD` message from the controller. This does not mean that this reported violation has to be harmful and leads to a non-deterministic behavior of the network, just that one can not predict which of these packets, the `FLOW_MOD` ⑤ or `PACKET_IN` ⑩, is processed first. For the case that ⑩ is processed before ⑤, the switch sends it to the controller. In this case, an error-free controller detects that it already sent the `FLOW_MOD` messages to the switch and buffers the packet until ⑤ is processed by the switch. After this, it can send the packet back to the switch without any further `FLOW_MOD` messages.

After the pre-processing stage, we have a set of violation graphs. Each of them contains exactly one violation and all events that led to the violation. This set of graph is the input for the next stage, the clustering.

4. Clustering

In this section, we describe BigBugs clustering algorithm. BigBug first relies on graph isomorphism to initialize the set of clusters (section 4.1). Then, it refines those by grouping related (but not equivalent) violations according to the SDN-specific features they share (section 4.2). For this, BigBug relies on a distance metric (section 4.3). We describe the full clustering algorithm in section 4.4. Finally, in section 4.5 we present different approaches we discarded throughout the implementation of BigBug.

4.1. Cluster initialization

BigBug first clusters each violation according to an isomorphism check, essentially grouping together violations containing equivalent event sequences.

In BigBug, we restrict the notion of event equivalence to the event type, not the actual content of the event. Specifically, we say that two violation graphs G and H are isomorphic (and therefore grouped in the same cluster) *if* each node in G can be exactly mapped to a node in H with the same event type and the same set of edges.

While checking for graph isomorphism can be done in quasipolynomial time [17], it can still take a long time to complete in practice. Therefore, we added a timeout of 10 seconds for the computation that checks if two graphs are isomorphic. If the timeout is hit, two graphs are considered *not isomorphic* and put in different clusters. Note, that they can still be clustered together in later stages.

Example To illustrate the isomorphic cluster initialization, we expand our example of the load-balancing application. Assume an additional host (H2) that sends a Web request analogous to H1. The extracted graphs for the additional violations look exactly the same as the previous ones, but with different event content, e.g. different packets, different hosts and depending on the load balancing different replicas. Figure 7a depicts the two known graphs from Figure 6 along with two new graphs. The isomorphic initialization correctly clusters the two pairs of isomorphic graphs as shown in the figure.

Figure 7b shows two additional violations, that are not isomorphic, but originate from the same bug: the missing synchronization barrier between the FLOW_MOD and the PACKET_OUT messages from our example. The two graphs in the figure basically show the events that happen after the first *packet bounce* between switch and controller: The controller assumes that this is a new request and sends three new messages back to the switch. As shown in the figure, the additional messages cause new concurrency violations between them, but also concurrency violations with the previous messages caused by the first PACKET_IN. These violations serve as an example of why the isomorphic initialization by itself is not enough to cluster violations caused by the same bug.

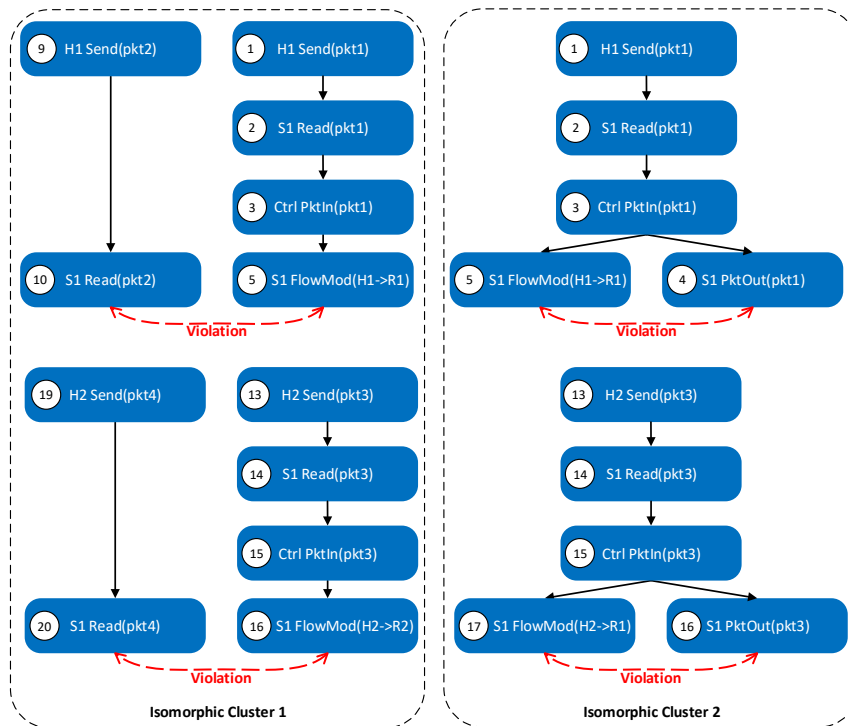


Figure 7a: The graphs in the two clusters are caused by the same root cause and clustered by the isomorphic initialization.

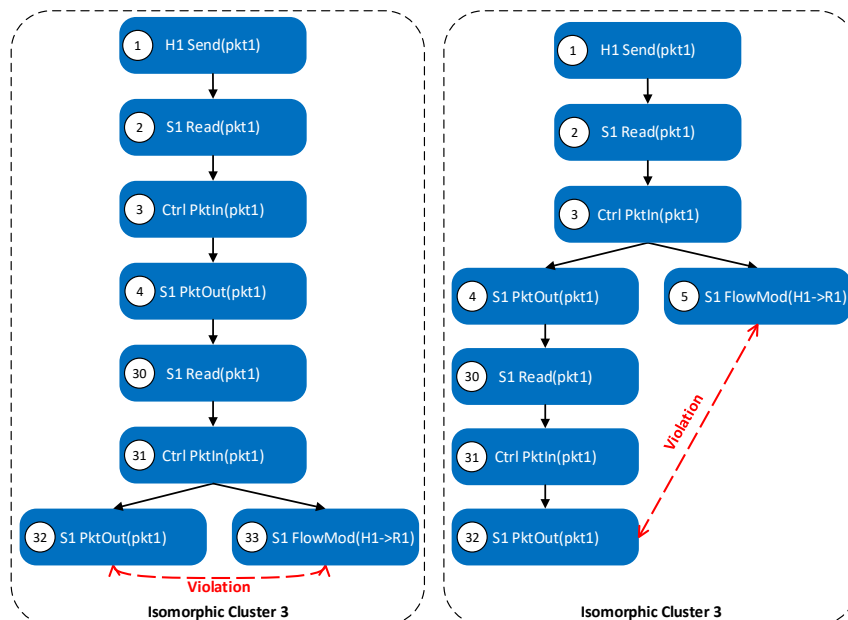


Figure 7b: Example of two graphs that are caused by the same bug, but are not captured by the isomorphic cluster initialization step, and thus, show that the isomorphic initialization by itself is not sufficient.

4.2. Identifying related violations through SDN-specific features

As already mentioned in the overview (chapter 2), the isomorphic cluster initialization successfully clusters “look-alike” violation graphs, but already fails to cluster slightly different graphs. Therefore, **BigBug** detects domain-specific features computed over each graph to calculate a distance between clusters. The distance between any two initialized clusters is used to build a distance matrix which is used to refine the initial clustering, clustering together closely related (but not equivalent) violations.

BigBug uses two different feature types: (i) boolean features that either exist or not in a violation graph, *e.g.* the graph has a packet flooding event; and (ii) numerical features that represent how often the feature is present in the graph, *e.g.* the number of host send events in a graph.

Formally, let G_k be the set of graphs in cluster C_k and $F_i : G_k \rightarrow \mathbb{N}$ be a function that returns the number computed for feature i . If feature i is boolean, F_i returns 1 if a graph has this feature, 0 otherwise. If feature i is numerical, F_i returns the actual number of occurrences of the feature.

We now present the seven different features currently implemented in **BigBug**.

1. **PACKET_IN/PACKET_OUT bounce** This boolean feature captures repeated `PACKET_IN` and `PACKET_OUT` events between the controller and a given switch for the same given packet. This situation occurs when the controller does not use proper synchronization primitives to ensure the rule that matches the packet has been committed to the Flow Table before sending the `PACKET_OUT` back to the switch (see section 2.1).
2. **Reply packets** This boolean feature captures if the violation was triggered by a host replying to a packet that it received. In many cases, the controller installs the forward path and the return path for a flow at the same time. The intuition behind this feature is to consider graphs with concurrency violations on either path closer to each other.
3. **Flow expiry** OpenFlow allows flow entries to expire after a certain specified (hard or soft) timeout [2]. While soft timeout helps cleaning the flow table, defining the timeout is usually tricky in asynchronous environments. Often, early flow expiry leads to many concurrency violations. This boolean feature captures violations caused by a flow expiry event.
4. **Flooding** Often, controllers flood packets for various reasons; *i.e.*, the controller discovering the network topology or it is not aware of the location of the destination host of the packet. However, the paths and the event ordering that follows a packet flood is completely nondeterministic. If miss-handled, flooded packets cause concurrency violations. The corresponding graphs are often completely different. As such, this boolean feature simply captures if packet flooding caused the violation.
5. **Number of root events** This numerical feature returns the number of root events in the violation graph. A root event is an event with only outgoing edges in the violation graph. The number of root events indicates if the violation is caused by just a single root event or by multiple ones.

6. **Number of host sends** This numerical feature, similar to the previous one, returns the number of host send events in the violation graph.
7. **Number of proactive violation events** SDNRacer distinguishes two types of events: reactive and proactive. Reactive events are the ones sent by the controller in response to received messages, while proactive events are sent independently. This numerical feature returns the number of events involved in the concurrency violation which are proactive, i.e. zero, one or two.

Our experiments (section 6) and manual analysis of various HB-graphs indicated that not all features carry the same significance in relating two violations. For instance, violations sharing the *flooding* feature tend to be more related than violations sharing the feature *reply packets*. We capture this effect in the distance function (section 4.3) by assigning different weights to each feature.

Example Consider the violation graphs from the cluster initialization in Figures 7a and 7b. Table 1 displays the features for the graphs in the four initialized clusters, as the graphs in the clusters have the same features.

	Bounce	Reply Pkt	Flow Exp.	Flood.	# Roots	# Host Sends	# Proactive
Cluster 1	0	0	0	0	2	2	0
Cluster 2	0	0	0	0	1	1	0
Cluster 3	1	0	0	0	1	1	0
Cluster 4	1	0	0	0	1	1	0

Table 1: Features of the four clusters after the isomorphic initialization in Figures 7a and 7b.

The graphs in cluster 1 have 2 root events each, e.g ① and ⑨ in the upper graph, that both are host send events. Therefore, the two functions for the features *number of root events* and *number of host sends* both return 2. All other features are not contained in these two graphs, so the respective functions return 0 for them. Similar to the first cluster, the graphs in the second cluster have the same features, except that they both only have one root event, a host send. The graph in cluster 3 has exactly the same features as the graph in cluster 4. They both have one root event that is a host send and do share the *PACKET_IN/PACKET_OUT bounce* feature. All other features are not exhibited in these graphs, and therefore 0.

Adding features Even though the feature set presented in this section provides a sound clustering (see chapter 6), it still is a heuristic approach. We are aware, that there is no guaranteed relation between the root causes of any two violations, even if the corresponding graphs have exactly the same features. Further, it might be necessary to implement additional features for applications or controllers which we did not use for the implementation and evaluation of BigBug. Therefore, we made it easy to add new features to improve the clustering for those cases. To add features, one needs to do the following:

1. Detect the feature in the violation graphs (see section 4.2)

2. Provide a function that uses this feature to calculate a distances between 0 and 1 between any two cluster (see section 4.3)
3. Assign a weight to this feature (see section 4.3)

4.3. Distance calculation

After **BigBug** extracts the features of each graph in a given cluster, it computes the mean of each feature in the cluster. Even though the calculation of the mean is the same for numerical and boolean features, the meaning of it differs. While it means the average number of occurrences of the feature in the graph for numerical features, it can be seen as a percentage of graphs that have the feature for boolean features.

Let $\{g_1, \dots, g_n\} \in G_k$ be the set of graphs in cluster C_k . The mean of feature i is computed as:

$$m_i^k = \frac{\sum_{l=1}^{|G_k|} F_i(g_l)}{|G_k|}$$

Observe, that with the currently implemented isomorphic initialization step, all graphs in an initialized cluster share exactly the same features, which makes the above calculation redundant. Still, there is a good reasons to keep it this way: as mentioned before, it might be necessary to add new feature, and this new features might not be the same for isomorphic graphs. We want **BigBug** to be extendable without the restriction that features have to be consistent for isomorphic graphs, and thus use this calculation. Further, with this approach, we also allow different initialization methods, not only the currently used isomorphic initialization, as the feature based clustering is independent of the cluster initialization method.

After the calculation of the mean for each cluster, our distance calculation algorithm computes the distance between any two clusters per-feature. The computation for each feature returns a value between 0 and 1 for both feature types, but the algorithm treats boolean and numerical features differently:

Boolean features The main idea behind the distance calculation for boolean features is to put clusters that have similar percentages of graphs with a certain feature closer together. For instance, if all graphs of a cluster have the *flooding* feature but only a few of the graphs of another cluster have it, the distance between these clusters should be bigger then for two clusters whit most of the graphs having the flooding feature.

The calculation for the per-feature distance between two clusters C_l and C_k for boolean feature i is:

$$d_i(C_l, C_k) = abs(m_i^l - m_i^k)$$

Numeric features With the per-feature distance calculation for numeric features, we want to put graphs with the same number of occurrences to be close together. For example, two clusters that contain only one host send per graph are related regarding

the feature *number of host sends*, therefore the distance between them should be small. On the other hand, even a small difference in the average number of occurrences of a feature between two clusters can mean that they are completely unrelated. Therefore, we only differentiate between equal and unequal average number of occurrences in two clusters.

The calculation for the per-feature distance between two clusters C_l and C_k for numeric feature i is:

$$d_i(C_l, C_k) = \begin{cases} 0 & \text{if } m_i^l = m_i^k \\ 1 & \text{if } m_i^l \neq m_i^k \end{cases}$$

Total distance As mentioned in section 4.2, we use weights according to the significance of the features. The total distance between any two clusters C_l and C_k regarding all j features is therefore:

$$d(C_l, C_k) = \sum_{i=1}^j w_i d_i(C_l, C_k)$$

BigBug computes the distance between any two initialized clusters. Then, it uses this distances to build a distance matrix which is then used by to the clustering algorithm.

Example Again, consider our example with the four initialized clusters with the features in Table 1 and assume an equal weight of 1 for each feature. Table 2 shows the resulting distance matrix.

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Cluster 1	0	2	3	3
Cluster 2	2	0	1	1
Cluster 3	3	1	0	0
Cluster 4	3	1	0	0

Table 2: Distance matrix calculated from the cluster features in table 1, assuming equal weights of 1 for each feature.

4.4. Clustering algorithm

We now describe **BigBugs** complete clustering algorithm. The four major steps of the whole clustering algorithm are:

1. Initialize the clusters using the isomorphic check (section 4.1)
2. Evaluate all the feature-based pair-wise distances (section 4.2 and 4.3)
3. Construct a distance matrix using distances values (section 4.3)
4. Agglomerative clustering

The first three steps, were already explained in the previous sections. They provide the basis for the fourth step, the *agglomerative clustering*, a variant of hierarchical clustering. In general, hierarchical clustering [18] builds a hierarchy of clusters, either bottom-up (agglomerative) or top-down (divisive). The agglomerative clustering starts with each element in an own cluster and recursively merges the closest clusters to new ones, until only one cluster is left. Divisive clustering does exactly the opposite: it starts with a single cluster and recursively splits it until each element is in its own cluster. We decided to use agglomerative clustering for **BigBug** as there is already an implementation of it in the SciPy hierarchical clustering package [19]. Specifically, we used the two function `linkage` and `fcluster`.

The *agglomerative clustering* algorithm takes the initialized clusters and the previously calculated distance matrix as inputs. The algorithm treats the initialized clusters as single elements and starts with each element in a separate cluster. Then, it performs the following recursively:

1. Find the smallest distance between any two clusters
2. Merge all clusters with this minimum distance between them into new clusters
3. Calculate the distance from each new cluster to any other cluster
4. Update the distance matrix

The first two steps are straight-forward: the algorithm selects all clusters with minimum distance to each other and merges them. The third step needs a more detailed explanation: To calculate the distance between a newly merged cluster and any other cluster, we use the linkage criterion *complete linkage*. This means, the distance between the new cluster and any other cluster is the maximum distance from any of its elements to the other cluster. Formally, let C be the set of all clusters and C_{new} be the cluster formed out of the clusters C_i . C_p denotes any clusters which is element of C but not C_{new} . The distance between C_{new} and C_p is calculated as follows:

$$d(C_{new}, C_p) = \max\{d(C_i, C_p) : \forall C_i \in C_{new}\}$$

After the algorithm calculated all distances from the new cluster to all other clusters, it updates the distance matrix with the new values. To do so, it removes the clusters that were merged and adds the new clusters instead. Then it repeats this four steps.

As a clustering until only one cluster is left is pointless, we use a stopping criterion for the agglomerative clustering. To do so, we define a *maximum distance* that we allow between any elements in a cluster. The algorithm stops as soon as the next merging step would result in a new cluster in which the distance between any two elements is more than the *maximum distance*.

Example Consider the distance matrix in Table 2. In the first step, the algorithm finds the minimum distance of 0 between any clusters, i.e. the distance between clusters 3 and 4. Therefore, the two clusters are merged into a new one, lets say cluster 5. Then, the algorithm calculates the distance between the new cluster and the remaining clusters,

and updates the distance matrix accordingly. As said, the distance between a new cluster C_{NEW} and another clusters C_p is defined as the maximum distance between any of its elements and C_p . The resulting distance matrix is shown in table 3.

	Cluster 1	Cluster 2	Cluster 5
Cluster 1	0	2	3
Cluster 2	2	0	1
Cluster 5	3	1	0

Table 3: Updated distance matrix from Table 2 after first merging step of the agglomerative clustering algorithm.

In the second iteration of the agglomerative clustering, the algorithm detects 1 as the minimum distance between any clusters, and therefore, merges clusters 2 and 5 to cluster 6. The new distance between cluster 6 and the remaining cluster is 3 as it is the maximum distance between cluster 1 and the merged clusters 2 and 5.

The third iteration would simply merge the last two cluster 1 and 6 and we would end up with a single cluster containing all graphs. As this would be pointless, we need to find an appropriate value for the *maximum distance* in the stopping criterion. Generally, the distance has to be in the range of 0 to the sum of the feature weights, as the sum of the weight equals the maximum achievable distance between any two clusters. In our example, the weights are all 1, and therefore, the *maximum distance* should be in the range 0 to 7.

If we choose the *maximum distance* to be 0, the clustering stops right after the first merge operation and we end up with three final clusters. Even tough this results in a sub-optimal clustering, as the violations in the clusters 2,3 and 4 are all caused by the same bug, the missing synchronization barrier, it still already cuts the expense for the developer in half as it reduces the six violations to 3 clusters. In reality, this would be much more, as there are thousands of violations in the trace, and a lot of them share the features in the examples.

Still, if we choose the *maximum distance* to be 1 or 2, the merging in the second iteration of the algorithm takes place and the algorithm therefore clusters the violation that origin from same bug correctly. We end up with two clusters, which is the optimum for this example.

A *maximum distance* of 3 or higher result in only one cluster, which is not appropriate, as the graphs in cluster 1 have a different root cause than graphs in cluster 6.

Even tough coming up with appropriate weights and maximum distance seems simple in this example, it is normally a difficult task, as we want **BigBug** to provide a solid clustering for different applications and controllers, that have different bugs. As the violation graph from different bugs have different appearances, finding the right values is a trace-off between accurate clustering and support of multiple controllers and applications.

4.5. Discarded approaches

In the process of developing **BigBug**, we tested different distance metrics and clustering algorithms, which we later discarded for different reasons. In this section, we present

two clustering methods (section 4.5.1), as well as two additional comparison methods for clusters (section 4.5.2), that did not make it into the final version of **BigBug**.

Apart from the actual clustering and the distance calculation, the processing pipeline for the clustering was always the same as in the final version described in this chapter. We start with the isomorphic initialization of the clusters, providing us with the elements for the actual clustering. Then, we calculate the pair-wise distances between all these clusters and build a distance matrix. Finally, we run the clustering algorithm.

4.5.1. Discarded clustering algorithms

In the following, we present the two clustering algorithms *k-medoids* [20] and *DBScan* [21], together with the reasons why we did not use them in the final version of **BigBug**.

k-medoids k-medoids is a variant of the k-means clustering algorithm. The idea behind k-means is to partition elements into k clusters, where each element is put in the cluster with the closest mean. Since, in our case, we only defined a pair-wise distance between graphs and clusters, we can not calculate an absolute cluster mean, and therefore, have to use k-medoids. k-medoids follows the same principle as k-means but does choose actual element as cluster centers (i.e., medoids) instead of the mathematical mean of a cluster. The medoid of a cluster is defined as the element with the smallest sum of distances to all other elements in the cluster. Further, the total cost of a clustering is defined as the sum of the distance from each element to its respective cluster medoid. To start the algorithm, it is necessary to first initialize k medoids. We choose the medoids for the first iteration to be the k isomorphic initialized cluster with maximum distance in between them. Another approach we tested was to randomly initialize the cluster medoids, but the results generated with the previously explained initialization were better. The k-medians algorithm in pseudo code can be found in Listing 1. Note that the single elements for the algorithm are the isomorphic initialized clusters, not single violation graphs.

Listing 1: k-medoids algorithm

```

1 initialize medoids
2
3 while cost decreases:
4     for each element not in medoids:
5         assign e to closest medoid
6
7     for each cluster C:
8         calculate medoid of C
9
10    calculate cost of clustering

```

This method provided accurate results for k close to the number of bugs, but as the number of bugs is normally not known previously this algorithm was not suited for our purpose.

DBScan DBScan is a density based clustering method. It aims to cluster elements in high-density areas (elements with many close neighbors) and marks elements that are far apart from others as outliers. It has two parameters, a minimum number of elements (*eMin*) and a radius (*r*).

An element e is marked as core element if it has at least $eMin$ elements in its neighborhood with distance r or smaller (r -neighborhood). All elements in the r -neighborhood of e are reachable from e . Further, if an element p is reachable from e and p is a core element, then an element q is reachable from e if q is reachable from p . Note that reachability is not symmetric: p is only reachable from q if q is a core element as well. All elements that are not reachable from any core element are marked as outliers. The DB-Scan algorithm is depicted in Listing 2. Note that the single elements for the algorithm are the isomorphic initialized clusters, not single violation graphs. Further, an element e that is first marked as an outlier, can later be added to a cluster if it is reachable from one of its core elements.

Listing 2: DBScan algorithm

```

1  for each element  $e$ :
2      if  $e$  was already visited:
3          continue
4      else:
5          mark  $e$  as visited
6
7      if  $e$  is not core element:
8          mark  $e$  as outlier
9      else:
10         create new cluster  $C$ 
11         add all elements reachable from  $e$  to  $C$ 

```

Even with multiple different values for $eMin$ and r , we could not get a good clustering for most cases. Either we had clusters with big distances between the single graphs in them or we ended up with far too many clusters. Therefore, we decided to discard this algorithm and use the agglomerative clustering instead.

4.5.2. Discarded distance metrics

In addition to the feature-based distance functions explained in sections 4.2 and 4.3, we utilized two other methods to measure the similarity between clusters. These methods are not feature-based, but also return a value between 0 and 1, and can, therefore, be added in the weighted sum of the distance calculation formula (section 4.3). The two methods are *isomorphic components* and *common write events*.

Isomorphic components To explain this violation graph comparison method, we first have to provide some more information about the violation graph structure. As already explained in section 3.2, we extract the per-violation graphs by upward traversal of the HB-graph, starting from the two violation events. All of these graphs have exactly two leave nodes, the violation events and a variable number of events that led to the violation. Further, the violation graphs can either consist of one or two weakly connected components [22]. A weakly connected component is a part of a graph, where each node is weakly connected to all others in the same part. Weakly connected means connected regardless of the edge direction (consider the edges as undirected). For our case, a graph either consists of only one component, if and only if the two violation events are weakly connected, or of two components, if not.

Figure 8 shows two generic violation graphs. On the left side, the violation graph

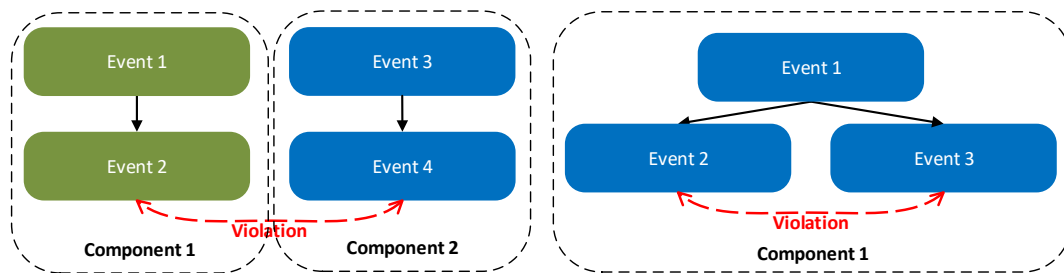


Figure 8: Example of two generic violation graph. The left one consists of two weakly connected components, the other only has one.

consists of two components, whereas the graph on the right side has only one. Note that the “violation marker” (red dotted edge) is not an actual edge of the graph, but only to show which events take part in the violation.

The idea behind the function *isomorphic components* was to extend the isomorphic initialization for parts of violation graphs. While in the initialization we restrict the isomorphism to the whole graph, we check not connected components of the graphs in this functions. Specifically, we do the following: *first*, we check if each of the two violation graphs we want to compare consist of two weakly connected components. If there is only one connected component in one of the graphs, there is no need to proceed this function, as we already checked the whole graph for isomorphism in the initialization. The function returns 1 in this case, meaning the graphs are not related in terms of isomorphic components. *Second*, we build violation subgraphs out of the two separate components of both graphs. *Third*, we check if a component of one graph is isomorphic to a component of the other graph. Here, we only consider components that contain a `FLOW_MOD`, i.e. write event. This, because else we would consider components with only a “basic read”, i.e. a read event like in the example of the unfixable violation shown on the left side of Figure 6 (green events), even though this read can violate concurrency with any arbitrary write event. Summarized, the function consider violation graphs close to each other if the have isomorphic components that contain a `FLOW_MOD`.

We explained the comparison of two single graphs above, not clusters as in the distance calculation, but since we use the isomorphic initialization all graphs in a cluster have the same structure and we can take an arbitrary graph out of each cluster as input for this function.

Even though this function helped to find related graphs in the first versions of `BigBug`, it also added another computational expensive check for isomorphism. Further, with the final domain-specific feature set we described in section 4.2 the benefit was negligible. Therefore, we discarded this function.

Common write events Apart from *isomorphic components*, we also tested a function we called *common write events*. The idea behind this function is that violations that share a violation event — exactly the same, not only the same type — have something in common. Again, as in the *isomorphic components* function, we only consider `FLOW_MOD` violation events in the check, as the read events can be arbitrary. The function basically

returns a value between 0 and 1 representing a percentage of graphs of one cluster that do not share a `FLOW_MOD` event with at least one graph of the other cluster.

As with the isomorphic components function, *common write events* did not provide significant benefits when tested with the final set of features. Thus, we discarded this function as well.

The clustering explained in this chapter successfully groups concurrency violations with the same root cause. Still, it is necessary to find the graph in each clusters that best represents it. The next chapter presents our ranking function.

5. Ranking

While the clustering algorithm groups the concurrency violations into a small number of clusters, the number of violations per cluster can be large; potentially in the order of 1000s. Our ranking function selects the most representative violation for each cluster.

The main intuition is to find the smallest graph that exhibits the most common features across all graphs in one cluster. The ranking function starts with examining the boolean features first. It selects all violation graphs that have all the boolean features exhibited in 50% or more of the reported violations in the cluster. The second stage is to reduce the set of chosen graphs based on the numerical features, one by one. Generally, we chose the graphs with the minimum difference between the feature in the given graph and the overall mean for the cluster. The order of selecting the graphs based on the numerical features is: the number of proactive violation events, the number of host sends, and finally, the number of root events. For the final set of chosen graphs, our ranking function selects the graph with the minimum number of events to present to the controller developer, as we think a smaller graph is easier to understand for the developer. If all of them have the same size, the ranking randomly picks one of them.

In summary, the ranking works as follows for each cluster:

1. Find the graphs that have all boolean features, that appear in minimum 50% of all graphs of the cluster, but none of the others, and mark them as candidates.
2. Process the numerical features sequentially in the order: proactive violation events, host sends and finally root events. For each of the numeric features do:
 - a) Find the graph with minimum distance in the feature to the cluster mean
 - b) Remove all graphs that have a higher distance from the candidates
3. From the final set of candidates, choose the smallest graph as the representative graph. If there are multiple smallest graphs in this set, choose one randomly.

	Bounce	Reply Pkt	Flow Exp.	Flood	# Roots	# Host Sends	# Proactive
Graph 1	0	0	0	0	1	1	0
Graph 2	0	0	0	0	1	1	0
Graph 3	1	0	0	0	1	1	0
Graph 4	1	0	0	0	1	1	0
Cluster	0.5	0	0	0	1	1	0

Table 4: Features of each graph in Figure 9 separately and average of the whole cluster.

Example Figure 9 depicts one of the two final clusters from the clustering example in chapter 4. For each of the graphs in the figure, table 4 shows its features as well as the average in cluster.

The ranking function first considers the boolean features only (first four columns). As there are 50% of graphs in the cluster that have the *PACKET_IN/PACKET_OUT bounce* feature, the ranking considers the graph 3 and 4 as potential representative graphs. Then, the ranking considers the numeric features in the order *the number of proactive violation events, the number of host sends, the number of root events*. For each of the numeric features, it chooses the graphs among the candidates that have minimum distance to the cluster mean. As both, graph 3 and 4, have the same numeric features, they both still are in the set of potential representative graphs. In the last step, the ranking chooses the smallest of the remaining graph. Still, graph 3 and 4 have the same number of events, and therefore, the ranking randomly picks one of them and outputs it as the representative graph of the cluster.

The ranking was the last stage of the processing pipeline of BigBug. First, the pre-processing trims the HB-graph and extract per-violation sub-graphs of the HB-graph. Then, the clustering starts with an isomorphic cluster initialization. After that, domain-specific features are used to define a distance metric between clusters, which is then used for the agglomerative clustering. The ranking function selects the most representative graph for each of the final clusters. In the next chapter we present the evaluation of BigBug.

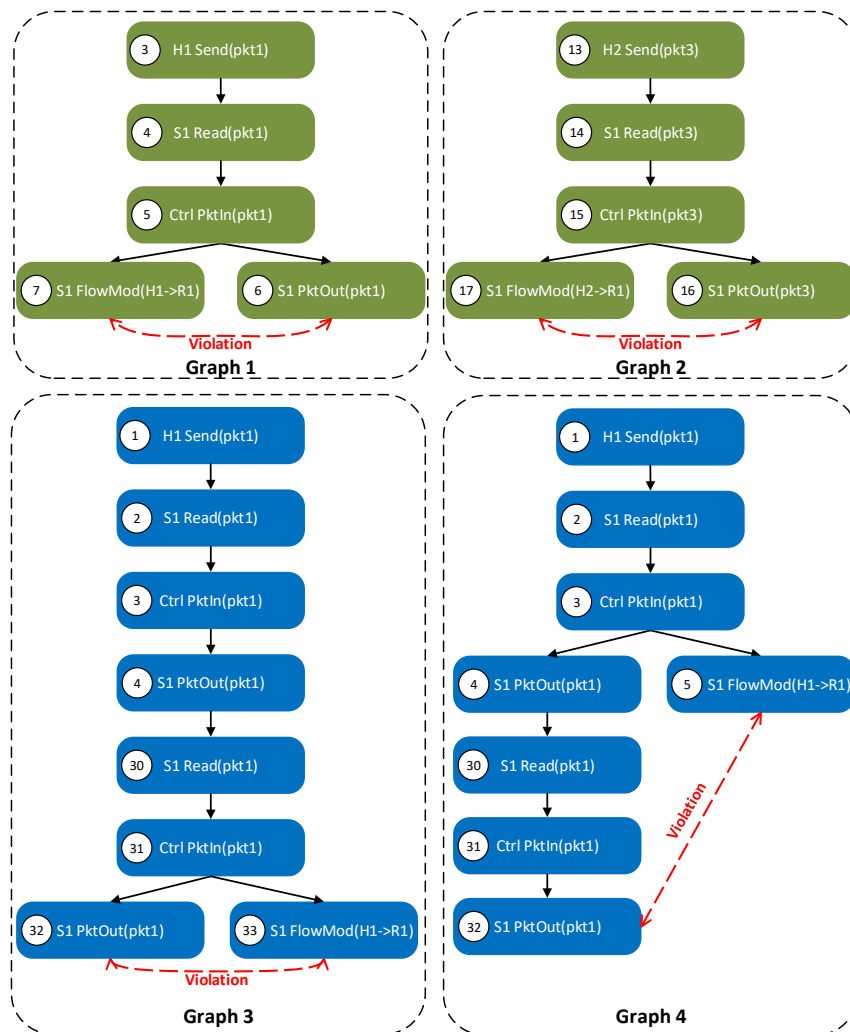


Figure 9: Example of graphs in a final cluster. The ranking function selects the most representative graph.

6. Evaluation

We implemented a working prototype of **BigBug** and used it to filter out the output produced by **SDNRacer**. Then, we evaluate it on multiple different applications of different controllers on a variety of topologies to show its usability. Further, we present a real world example where we used **BigBug** to fix bugs in an application. The results show, that **BigBug** is of great use in finding representative violations, that hold enough information to fix them and when fixed, reduced the number of violations by more than 99%. Additionally, we evaluate the performance and show that **BigBug** finishes analyzing traces within one second for 60% of the traces.

In this chapter, we show the experimental setup in section 6.1. We analyze the usability and provide a real-world use case in sections 6.2 and 6.3. Then, we evaluate the influence of the simulations length on the number of representative graphs reported by **BigBug** in section 6.4. Finally, we evaluate the performance of **BigBug** in section 6.5.

6.1. Experimental setup

We used **BigBug** to find violations produced by actual SDN controllers on multiple network topologies. Each simulation configuration was ran for 200, 400, 600, 800 and 1000 simulation steps and repeated 15 times to provide more reliable results. All simulations were executed on a laptop with an Intel Core i7-3740QM CPU running at 2.70GHz with 16 GB RAM. These traces where then copied to a server with two Intel Xeon E5-2670 CPUs running at 2.30GHz and 128GB of RAM to run **BigBug** on them.

We mainly report the number of violations for Floodlight v0.91 [6] and POX EEL [23]. Additionally, we also fixed the concurrency issues in the Floodlight Load Balancer module (Floodlight Fx) so as to provide a real-world use case. Further, we used the fixed POX EEL Forwarding module (POX EEL Fx) that was fixed in the development process of **SDNRacer**. We used the following applications shipped with these controllers for the evaluation:

Admission Control (Floodlight [24]): This application is used to allow or block communication between hosts in a network. Whether two hosts are allowed to communicate, is controlled by ACL rules (Access Control List) with different priorities.

Circuit Pusher (Floodlight [25]): The Circuit Pusher proactively installs permanent routes between hosts in the network. Routes can be added by the network administrator or programs via Floodlights rest API.

Forwarding (Floodlight [26], POX [27], POX Fx): This applications builds and maintains a network wide MAC address table, as opposed to the per-switch tables used by

common L2 Learning Switch applications. Flow rules are added once the controller learns about a new device.

Learning Switch (Floodlight [28], POX [29]): Common L2 Learning Switch application. Builds and maintains a MAC address table for each switch separately, storing the physical port on which each device can be reached.

Load Balancer (Floodlight [7], Floodlight FX): This applications balances load (ping, TCP and UDP flows) on a set of replicas behind a virtual IP address. Every time a new packet is sent to the virtual IP, the application selects a replica and installs flow rules for the entire path through the network.

As different network topologies can significantly influence the results, we generated traces on three different network topologies:

Single A single switch connected to two hosts¹.

Linear Two hosts connected via two switches.

Binary tree Seven switches in a binary tree topology with four hosts connected to each leaf switch.

As explained in section 4.2, we assign a weight to each feature in the distance calculation. A simple sensitivity analysis, where we tested different weights for each feature, led us to use the weights shown in Table 5 for the distance function (section 4.3). Further, we define the maximum distance between any to violation graphs in a cluster to be 2 (section 4.4). We are aware that different weights can result in an even better (or worse) clustering and leave a full sensitivity analysis for later work. Throughout all this experiments, we used the default configuration for **SDNRacer**.

Feature	Weight
PACKET_IN / PACKET_OUT Bounce	2.0
Packet Flood	2.0
Flow Expiry	2.0
Number of Proactive Violations	1.5
Number of Host Sends	1.0
Number of Root Events	0.5
Reply Packets	0.5

Table 5: Feature weights used for the evaluation of **BigBug**.

The above experiment configurations resulted in more than 100 different experiments that we repeated 15 times each, totalling in more than 2000 executions of **BigBug**. In the following sections, we use representative parts of these results to evaluate **BigBug**. A full table containing the results of all experiments can be found in appendix A. The

¹ We used four hosts for the Load Balancer module.

values reported in all this tables, the full table in the appendix and the ones used in the remainder of this chapter, are the median of the 15 repetitions we carried out for each configuration. The following legend explains the columns in the different result tables.

Legend:

App	Tested application
Topology	Network topology used for the simulation
Controller	Tested controller
Steps	Simulation length (number of simulation steps)
SDNRacer	Values reported by SDNRacer
<i>Events</i>	Number of events in the trace
<i>Violations</i>	Number of violations reported by SDNRacer
BigBug	Values reported by BigBug
<i>Isomorphic Clusters</i>	Number of clusters after the isomorphic initialization
<i>Timeouts</i>	Number of isomorphism checks that hit the timeout
<i>Final Clusters</i>	Final number of clusters after the agglomerative clustering
Cluster	Information about the size of the final clusters
<i>Median</i>	Median of the cluster size
<i>Max</i>	Maximum cluster size
Timing	Timing information
<i>Total</i>	Total execution time (SDNRacer + BigBug)
<i>SDNRacer</i>	Execution time of SDNRacer
<i>BigBug</i>	Execution time of BigBug

6.2. Usability

In this section, we evaluate the usability of **BigBug** on a fixed trace length of 200 steps. Table 6 shows the results for different applications and topologies. Each value in the table is the median of the 15 repetition we ran for each configuration.

As the results show, **BigBug** is able to reduce the number of reported violations by up to three orders of magnitude to a maximum of 6 final clusters throughout all experiments. Moreover, the results also show that the number of final clusters is not proportional to the number of concurrency violations, as traces with more than thousand violations produce more or less the same number of clusters as traces with less than ten violations. The number of final clusters is much more dependent on the variety of violation graphs that are caused by a single bug in an application and how these graphs are related by our domain-specific feature set than by the number of violations in the trace. With this, we

App	Topology	Controller	SDNRacer		BigBug			Clusters	
			Events	Violations	Isomorphic Clusters	Timeouts	Final Clusters	Median	Max
Adm. Ctrl.	BinTree	Floodlight	908	81	26 (32.10 %)	0 (0.00 %)	3 (3.70 %)	24	33
	Linear	Floodlight	287	17	11 (64.71 %)	0 (0.00 %)	3 (17.65 %)	5	8
	Single	Floodlight	160	11	3 (27.27 %)	0 (0.00 %)	1 (9.09 %)	7	10
CircuitPusher	BinTree	Floodlight	1017	39	6 (15.38 %)	0 (0.00 %)	2 (5.13 %)	19.5	32
	Linear	Floodlight	248	42	6 (14.29 %)	0 (0.00 %)	2 (4.76 %)	24	39
	Single	Floodlight	363	47	3 (6.38 %)	0 (0.00 %)	2 (4.26 %)	23.5	29
Forwarding	BinTree	Floodlight	3016	288	58 (20.14 %)	0 (0.00 %)	3 (1.04 %)	31	215
		POX EEL	5632	310	160 (51.61 %)	4 (2.08 %)	4 (1.29 %)	64.5	143
		POX EEL Fx	5419	52	48 (92.31 %)	0 (0.00 %)	4 (7.69 %)	7.5	32
	Linear	Floodlight	273	18	11 (61.11 %)	0 (0.00 %)	3 (16.67 %)	4	8
		POX EEL	389	13	12 (92.31 %)	0 (0.00 %)	3 (23.08 %)	4	7
		POX EEL Fx	347	6	6 (100.00 %)	0 (0.00 %)	3 (50.00 %)	2	2
	Single	Floodlight	423	10	5 (50.00 %)	0 (0.00 %)	2 (20.00 %)	7.5	9
		POX EEL	583	11	9 (81.82 %)	0 (0.00 %)	2 (18.18 %)	4	8
		POX EEL Fx	586	9	7 (77.78 %)	0 (0.00 %)	2 (22.22 %)	4.5	7
LearningSwitch	BinTree	Floodlight	6658	344	210 (61.05 %)	0 (0.00 %)	5 (1.45 %)	48	155
		POX EEL	3408	66	61 (92.42 %)	0 (0.00 %)	2 (3.03 %)	33	46
	Linear	Floodlight	228	15	12 (80.00 %)	0 (0.00 %)	2 (13.33 %)	12	14
		POX EEL	241	13	6 (46.15 %)	0 (0.00 %)	2 (15.38 %)	6.5	7
	Single	Floodlight	450	30	10 (33.33 %)	0 (0.00 %)	4 (13.33 %)	7	14
		POX EEL	372	6	3 (50.00 %)	0 (0.00 %)	1 (16.67 %)	6	6
LoadBalancer	BinTree	Floodlight	17593	1910	272 (14.24 %)	0 (0.00 %)	5 (0.26 %)	204	1362
		Floodlight Fx	7626	206	68 (33.01 %)	11 (7.48 %)	3 (1.46 %)	24	177
	Linear	Floodlight	2039	225	34 (15.11 %)	0 (0.00 %)	4 (1.78 %)	35	172
		Floodlight Fx	1341	19	6 (31.58 %)	0 (0.00 %)	3 (15.79 %)	7	10
	Single4	Floodlight	4034	1664	107 (6.43 %)	0 (0.00 %)	3 (0.18 %)	125	1529
		Floodlight Fx	1385	13	2 (15.38 %)	0 (0.00 %)	2 (15.38 %)	7	8

Table 6: Output of SDNRacer and BigBug on 200 step traces of different applications and topologies (median of 15 repetitions).

can assume that even for traces with much more concurrency violations, the developer will still only get confronted with a handful of representative violation graphs.

Figure 10 shows the CDF of the percentage of violations reduced by the clustering initialization and after the agglomerative clustering, for all traces lengths, not only 200 steps. The isomorphic cluster initialization already reduces the number of reported violations by more than 63% in 50% of all experiments. Further, the complete clustering process, the initialization followed by the feature-based agglomerative clustering, reduced the number of reported violations by more than 95% in 50% of all experiments, with a maximum reduction of 99.98%. The fact that **BigBug** reports a maximum of 6 representative graphs and the maximum number of violations in all experiments is more than 8000, highly supports the assumption, that the developer will only get confronted with a few graphs, even for traces with thousands of violations.

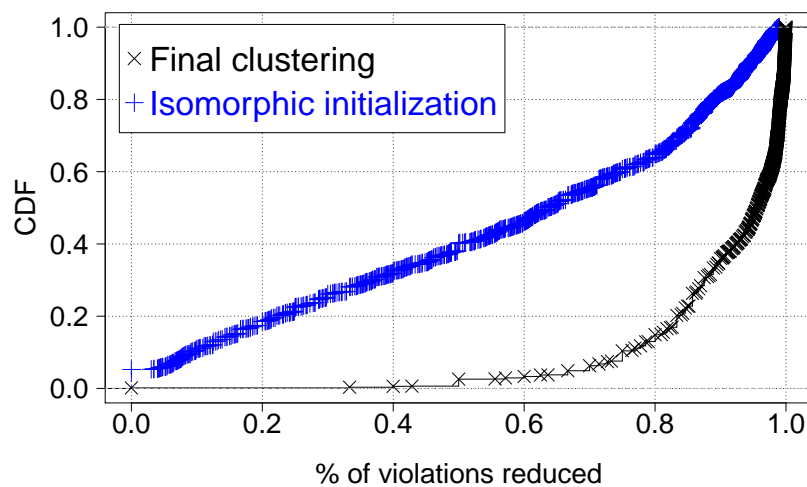


Figure 10: CDF of the % of violation reduction after the isomorphic initialization and the final clustering.

The last two columns in table 6 show the median and maximum cluster sizes. We can see, that the graphs are not uniformly distributed among the final clusters, indicating that some bugs in the controller produce much more concurrency violations than others. This can greatly help the developer, as it shows the quantitative impact of the underlying bug.

The timeout implemented for the graph isomorphism check triggered in only a small fraction of all experiments. Specifically, in the two POX EEL Forwarding modules (fixed² and original) and the two Floodlight Load Balancing modules (fixed and original²) on binary tree topology. This is due to the fact that the binary tree is by far the most complex topology out of the tested ones and these modules produce the biggest violations graphs. The POX EEL Forwarding modules use flooding, and therefore, generate violation graphs that span big parts of the HB-graph, sometimes containing thousands of events, which makes computations on them expensive. The original Floodlight Load Balancer module has the underlying bug that causes the packet bouncing, which can also

² In this modules, the timeout was only hit in traces longer than 200, and therefore it is not visible in table 6

results in huge graphs. In the fixed version of the Load Balancer, we got rid of the packet bouncing, but implemented synchronisation barriers. These barriers events can connect big parts of the HB-graph and therefore sometimes result in complex violation graphs. Even though the timeout is hit in a few cases, the results do not indicate that this has an influence on the final number of clusters.

6.3. Use case

Although the above results proof, that **BigBug** can greatly reduce the graphs shown to the developer, they do not indicate if the few reported graphs are useful for the developer and provide enough information to identify the bugs in the SDN application. Therefore, we show the usability of **BigBug** in a real-world example: we used the representative violation graphs reported by **BigBug** to fix the bugs in the Floodlight Load Balancer application. Specifically, we did the following:

1. Run **BigBug** on a trace of the Floodlight Load Balancer module on *single* topology with four hosts.
2. Use the representative graphs reported by **BigBug** to figure out what bugs are present in the application.
3. Fix the bugs
4. Repeat the experiment with the same configuration but the fixed version of the application.

BigBug reported three representative graphs for the Floodlight Load Balancer module. The first two graphs were similar to the example graphs from the load-balancing example (section 2.1), that showed an unfixable violation and a missing synchronization barrier. The third one indicated, that there is no buffering in the application. Figure 11 illustrates the third reported violation graph.

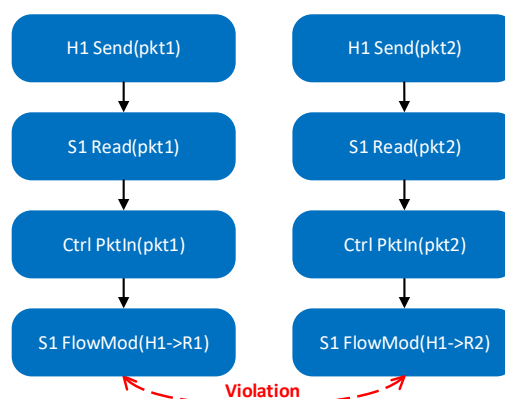


Figure 11: Illustration of one of the three representative graphs that **BigBug** reported for the Floodlight Load Balancer module.

Assume a host sends two similar packets right after each other and they both get dispatched to the controller. The original application processes these packets as follows. The controller first processes one packet, i.e. sends two `FLOW_MOD` and one `PACKET_OUT` messages. After that, the controller processes the second packet which again results in the same messages sent, but probably with a different replica as destination as the load-balancing application uses a round-robin fashion to assign the replicas. The graph in the figure provides enough information to figure out that the controller does not use proper buffering of packets, as it treats the second packet as if it is a new packet, unrelated to the first one.

Apart from this violation, the two `FLOW_MOD` messages that add the forwarding rule for the return path cause a violation as well. Further, the `PACKET_OUT` messages will also cause a violation with the `FLOW_MOD` messages in the figure. These violations are either part of the cluster with the missing synchronization barrier as root cause, or the one with the missing buffering.

App	Controller	Steps	SDNRacer		BigBug			Clusters	
			Events	Violations	Isomorphic Clusters	Timeouts	Final Clusters	Median	Max
LoadBalancer	Floodlight	200	4034	1664	107 (6.43 %)	0 (0.00 %)	3 (0.18 %)	125	1529
		400	11445	3281	253 (7.71 %)	0 (0.00 %)	3 (0.09 %)	228	3053
		600	21465	3577	280 (7.83 %)	0 (0.00 %)	3 (0.08 %)	239	3328
		800	23731	4161	281 (6.75 %)	0 (0.00 %)	3 (0.07 %)	253	3956
		1000	28600	6196	358 (5.78 %)	0 (0.00 %)	3 (0.05 %)	339	5882
	Floodlight Fx	200	1385	13	2 (15.38 %)	0 (0.00 %)	2 (15.38 %)	7	8
		400	3731	13	2 (15.38 %)	0 (0.00 %)	2 (15.38 %)	7	8
		600	6063	12	2 (16.67 %)	0 (0.00 %)	2 (16.67 %)	7	9
		800	8591	12	2 (16.67 %)	0 (0.00 %)	2 (16.67 %)	7	9
		1000	10475	14	2 (14.29 %)	0 (0.00 %)	2 (14.29 %)	7.5	9

Table 7: Comparison of original and fixed Floodlight Load Balancer module on single topology with four hosts.

We used this information to fix the two bugs, the missing barrier synchronization and the missing buffering, in the Load Balancer module. Table 7 shows the comparison between the original and fixed load-balancing applications for *single* topology. One can see, that the number of reported violations is reduced by two orders of magnitude for all trace lengths. Further, more than **99.76 %** disappeared in the fixed version on average for the 1000 steps trace. Out of the three reported violations by **BigBug** in the original version, only 2 are reported after we fixed the bugs. One of them obviously represents the same unfixable errors as in the original module. The additional cluster that **BigBug** reports for the fixed version is because we did not consider the case that a replica sends a packet to the host apart from responses to requests from the host. Thus, this reported violation still contains a graph indicating a missing buffering, but only for this specific case. The fact that the number of violations in the fixed module is about constant for variable trace lengths, indicates that the few violations happen right at the beginning of the trace and do not repeat themselves. This makes sense, as at some point, all possible routes between the hosts and replicas are added to the switch flow table. This was not the case for the original modules, as there are still packets bouncing between controller and switch even after a long time.

6.4. Influence of trace length on BigBug

The time needed to run the **SDNRacer** and **BigBug** can be long for long traces containing a lot of concurrency violations. Further, **SDNRacer** did not always successfully complete for huge traces with thousands of violations, even on the server with a huge amount of RAM and high computational power. Therefore, we examine the influence of the trace length on the output of **BigBug** in this section.

App	Topology	Controller	Steps	SDNRacer		BigBug		
				Events	Violations	Isomorphic Clusters	Timeouts	Final Clusters
CircuitPusher	Linear	Floodlight	200	248	42	6 (14.29 %)	0 (0.00 %)	2 (4.76 %)
			400	602	118	6 (5.08 %)	0 (0.00 %)	2 (1.69 %)
			600	947	179	6 (3.35 %)	0 (0.00 %)	2 (1.12 %)
			800	1280	258	6 (2.33 %)	0 (0.00 %)	2 (0.78 %)
			1000	1583	304	6 (1.97 %)	0 (0.00 %)	2 (0.66 %)
Forwarding	BinTree	POX EEL	200	5632	310	160 (51.61 %)	4 (2.08 %)	4 (1.29 %)
			400	9398	359	204 (56.82 %)	4 (2.02 %)	4 (1.11 %)
			600	12104	365	216 (59.18 %)	4 (2.05 %)	4 (1.10 %)
			800	14431	395	248 (62.78 %)	2 (1.05 %)	4 (1.01 %)
			1000	16860	402	234 (58.21 %)	2 (0.99 %)	4 (1.00 %)
LearningSwitch	Single	Floodlight	200	450	30	10 (33.33 %)	0 (0.00 %)	4 (13.33 %)
			400	1016	73	19 (26.03 %)	0 (0.00 %)	4 (5.48 %)
			600	1574	126	20 (15.87 %)	0 (0.00 %)	4 (3.17 %)
			800	2185	158	25 (15.82 %)	0 (0.00 %)	4 (2.53 %)
			1000	2733	208	28 (13.46 %)	0 (0.00 %)	4 (1.92 %)

Table 8: Influence of trace length on the number of final clusters reported by **BigBug** for different applications and topologies.

Table 8 shows the results for difference trace lengths, applications, and topologies. The results show that the number of final clusters is not affected by the trace length. Even if we consider all 15 repetitions separately, and not only the median like in this table, we never reported a difference of more than 1 cluster between the median and the extremas, in all our experiments. The variance in the final number of clusters was the highest in the experiments with less than 1000 events and much less than 100 concurrency violations in the trace. That is to be expected as a very low number of events and violations indicate a low network activity, and therefore, a low chance to trigger all bugs in the application.

While carrying out this part of the evaluation, we also discovered a flaw in **SDNRacer**. Specifically, in one of the filter functions that filter out harmless concurrency violations from the harmful: the time filter. The idea behind the time filter is that a potential concurrency violation is considered *not harmful*, and thus, is not reported as a violation, if two events happen more than certain time after each other, i.e. the chance that they are processed in the “wrong” order is close to 0. Even tough this is reasonable filter method, there is a discrepancy in the implementation. Although the trace is generated in a discrete-event simulation, the filter uses real-time time stamps of the violation events to calculate the time difference between them. This leads to a problem, because the processing time for a single simulation steps can greatly vary depending on how much happens in that single step. In fact, in simulations with high network activity it happens that two consecutive simulation steps are processed more than 2 seconds apart. Since the default setting for the time filter of **SDNRacer**, the setting we use in all experiments, is to filter out all violations in which the events are more than two seconds apart, **SD-**

NRacer will not report them even though they happen right after each other in simulation time. This problem causes the number of violations reported by SDNRacer to be highly dependent on the computational power used for the simulation.

Because of this flaw in SDNRacer, we sometimes got unexpected results for the Load Balancer module on binary tree topology. The number of reported violations for longer traces was sometimes lower than for shorter traces. A constant number of violations for different trace length is explainable, as there exist bugs that only trigger at the beginning of a simulation, but a decreasing number of violations indicates an issue within the system. As we already see the number of reported violations in the output of SDNRacer, this issue can not be located in BigBug, but has to be in SDNRacer. We believe that this only affects the quantity of the reported violations and not the quality, as the reported graphs still suffice to fix bugs in the controller, as section 6.3 showed.

Even with this, the results in the table still show, that the output of BigBug is almost unaffected by the trace length. Therefore, it is completely sufficient to run BigBug on shorter traces, and only increase the simulation length if no bugs are found. In fact, a trace length of only 200 steps was sufficient to find concurrency violations in all tested applications.

6.5. Performance

As we just showed, the trace length has no significant influence on the output of BigBug. Thus, we concentrate to evaluate BigBugs performance for a constant trace length of 200 steps. Figure 12 depicts the CDF of the execution time for BigBug for all experiments performed on 200 step traces. Even though the worst case run time of roughly 20 minutes is quite high, more than 80% of all experiments finish in less than 10 seconds, which is completely sufficient for BigBug to be of practical use.

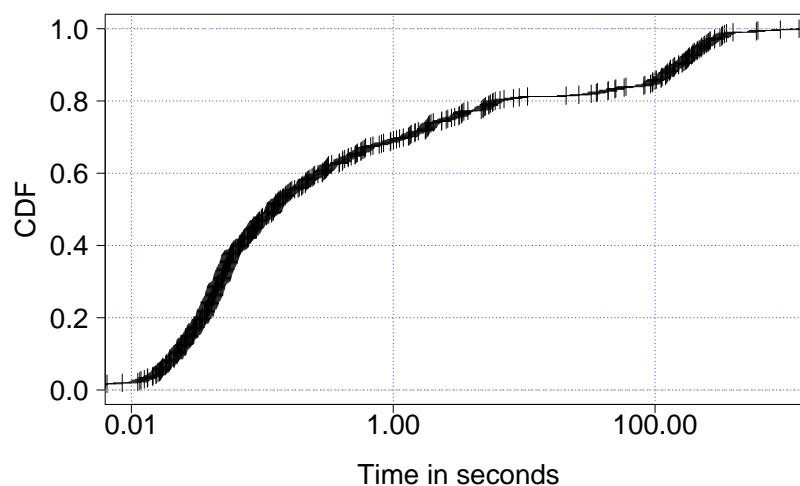


Figure 12: CDF of execution times for all simulation configurations with a trace lengths of 200 steps.

App	Topology	Controller	SDNRacer		BigBug		Timing		
			Events	Violations	Isomorphic Clusters	Final Clusters	Total	SDNRacer	BigBug
Forwarding	Bin Tree	Floodlight	3016	288	58 (20.14 %)	3 (1.04 %)	5.585 s	3.828 s	1.757 s
		POX EEL	5632	310	160 (51.61 %)	4 (1.29 %)	187.159 s	18.740 s	168.419 s
LearningSwitch	Bin Tree	Floodlight	6658	344	210 (61.05 %)	5 (1.45 %)	19.075 s	13.753 s	5.323 s
		POX EEL	3408	66	61 (92.42 %)	2 (3.03 %)	9.819 s	7.208 s	2.611 s
LoadBalancer	Bin Tree	Floodlight	17593	1910	272 (14.24 %)	5 (0.26 %)	326.107 s	116.644 s	209.463 s
		Floodlight Fx	7626	206	68 (33.01 %)	3 (1.46 %)	168.426 s	16.569 s	151.857 s
	Single4	Floodlight	4034	1664	107 (6.43 %)	3 (0.18 %)	146.332 s	29.504 s	116.828 s
		Floodlight Fx	1385	13	2 (15.38 %)	2 (15.38 %)	1.391 s	1.334 s	0.057 s

Table 9: Execution times for BigBug and SDNRacer for 200 step traces for different applications and topologies.

As it is necessary to first run SDNRacer before we can even use BigBug, table 9 shows the execution times of SDNRacer and BigBug on 200 step traces for different applications and topologies. It sticks out, that there are sometimes big differences between the execution times of BigBug and SDNRacer, and that they have different dependencies. In the following, we examine the two tools separately.

SDNRacer The run time of SDNRacer is highly dependent on the number of events in the trace, as for example the Load Balancer experiment on binary tree shows. This experiment has by far the highest number of events and also the highest execution time. Further, there is also a strong dependency on the number of violations in the trace, as the comparison between the Floodlight Load Balancer on *single* topology and the Floodlight Learning Switch on *binary tree* shows. Even though the trace for the Learning Switch has the higher number of events, SDNRacer still takes more time to finish on the Load Balancer trace, that has fewer events but about 5 times the number of violations.

These observations cover with our expectations, as SDNRacer first processes every single event in the trace. Then, SDNRacer proceeds by evaluating each combination of read and write events that have at least one write for potential violations.

BigBug The run time of BigBug on the other hand has different dependencies. The values in the table indicate a dependency on the number of violations in the trace, but not only, as the comparison between POX EEL Forwarding on binary tree and Floodlight Load Balancer on single topology show. Even though the trace for the Load Balancer contains much more violations, BigBug still has longer to finish on the Forwarding trace. Further, there also seems to be a relation between the execution time and the number of events. This is only partly true, as only the HB-graph size is related to the number of events, and only the pre-processing processes the whole graph. In the following we examine the different parts of the processing pipeline so as to find all dependencies between the run time of BigBug and the trace size and composition.

The first step is the pre-processing of the HB-graph, i.e. the trimming of the HB-graph and the extraction of per-violation graphs. The time it takes for the trimming is dependent on the size of the HB-graph, i.e. the number of events, but also on the structure and complexity of the graph. Even though this indicates a dependency on the number of events in the trace, an evaluation showed, that the trimming hardly ever contributes more than 1% to the total run time of BigBug, and thus its contribution to the total run

time can be neglected. The extraction of the per-violation graphs on the other hand has a strong dependency on the number of violations, as each of them is processed separately. Additionally, the size of the different violation graphs also contributes to the run time. In contrast to the trimming, this part can make up a huge part (>90%) of the total processing time in some experiments. Note, that we include the feature detection as part of the extraction, as the features are separately calculated over each violation graph right after the graphs were extracted.

In a closer examination of the clustering part, we discovered that only the isomorphic cluster initialization contributes significantly to the total run time of **BigBug**. Even though the number of executions of the distance calculation function is in the order of $O(n!)$ where n is the number of initialized clusters, the total time needed to process all these executions is insignificant, as it only consists of basic arithmetic operations. Once the distance matrix is built, the agglomerative clustering finishes in almost no time, and therefore is negligible as well.

The isomorphic cluster initialization does perform a lot of isomorphism checks. These checks are computationally expensive and can take a long time to finish. The number of checks is related to the number of violations in the graph, and even though we pre-sort the violation graphs based on their size and only check the graphs for isomorphism if they have the same size, there are still a lot of isomorphism checks involved for traces with a high number of violations. Further, the time needed for the isomorphism check is dependent on the size and complexity of the two graphs which have to be compared.

The ranking function is only called once for each final cluster, thus less than 7 times in all experiments. Further, there are no complex calculations to be made in this function. Thus, the time needed for this part is insignificant.

In summary, the execution time of **BigBug** is mostly dependent on the number of violations and their respective violation graph complexity. Even though there are dependencies on the number of events, the number of isomorphic clusters, and the complexity of the HB-graph, these do not significantly contribute to the total processing time of **BigBug**.

Out of this close examination of the two systems we state that primarily the number of events (**SDNRacer**) and the number and structure of the violation graphs (**SDNRacer** and **BigBug**) influence the run time. We believe, that with an appropriate simulation length, **BigBug** is fast enough for practical use. Further, once bugs in an application are fixed, there are less violations and therefore, the time needed to run **BigBug** is reduced, like the comparison between the fixed and original Load Balancer module show.

The evaluation in this chapter showed, that **BigBug** successfully reduces the number of reported violations to a handful of representative ones that hold enough information to identify and fix bugs in real-world applications. Further, short traces of 200 steps were sufficient to find violations in all tested applications and **BigBug** finished within a few seconds on the vast majority of these traces. Still, there is, as always, room for improvement, thus, we continue with suggestions for future work in the next chapter.

7. Outlook

The evaluation showed that **BigBug** is already of great use for a SDN developer, but there is still potential to improve it. In this chapter, we provide suggestions and approaches for future work on **BigBug**.

Although **SDNRacer** is not part of this work and only used as a tool to get the HB-graph and concurrency violations, we strongly recommend first fixing the issue with the time filter, as it leads to inconsistent results. To do so, we suggest to replace the real-time time stamp on the recorded events with a time stamp in simulation time, i.e. the simulation step in which the event was processed. After that, it is necessary to find an appropriate value for the number of steps between any events for the time filter to filter out violations, as it greatly impacts the number of reported violations.

The evaluation showed, that **BigBug** reports representative violations that can actually be used to fix bugs in real-world applications. Still, the number of controllers and applications we tested is limited. Even though we believe that our solution works for different controllers and applications as well, it is possible that one needs to add further features, or even add different cluster comparison methods, to successfully distinguish between different root causes of concurrency violations. In the following, we present a few ideas to further compare graphs:

Feature set comparison Consider the bug in the Load Balancer module of Floodlight with the missing synchronization barrier. A single host send that result in a `PACKET_IN` `PACKET_OUT` bounce between controller and switch already cause multiple concurrency violations as each bounce results in multiple messages send by the controller. These graphs are not isomorphic, but share some exactly same events, at very least the host send event, as well as the first `PACKET_IN` message sent to the controller. The idea of the feature set comparison is to check if the set of unique event IDs in two violation graphs are similar to each other, or one is even a subset of the other. With this, it would be possible to relate violation graph that origin from exactly the same events.

Exactly same root events Similar to the *feature set comparison* above, it could already be helpful to check whether two violation graphs origin from the exactly same events, not only the number of root events as we do in **BigBug**. This could also be applied for the *Reply packet* feature we use. Specifically, if two host handle events in two graphs are exactly the same.

BigBug already greatly reduces the number of graphs a developer has to examine, as the evaluation showed. However, it still can be hard to figure out the root cause of a violation graph, as they sometimes still contain many events. We believe that it is possible to further reduce the effort needed by the developer by simplifying the reported graphs. For example the substitution of multiple `PACKET_IN` / `PACKET_OUT` bounces between

a controller and a switch through a single node in the graph, or the replacement of a long data plane traversal of a packet without any controller interaction, could easily reduce the size of the graphs and improve the readability, without reducing its information content.

Additionally, **BigBug** could also be extended to provide suggestions on what root cause lays behind a concurrency violation. For example, if a `PACKET_IN` message that was sent to the controller is followed by a `PACKET_OUT` and a `FLOW_MOD` message without a synchronization barrier between them, the output of **BigBug** could be modified to suggest this missing synchronization barrier to the developer as possible root cause of the violation.

8. Conclusion

In this work, we introduced **BigBug**, a generic framework to automatically narrow down the most representative concurrency violations, i.e. the ones that best illustrate the likely root cause of an actual bug. To do so, **BigBug** processes the violations reported by concurrency analyzers in three steps:

1. The pre-processing trims the HB-graph and generates per-violation graphs as to examine them individually.
2. The clustering first initializes clusters with isomorphic violation graphs and then uses a distance metric, based on domain-specific features, for the agglomerative clustering that further narrows down the number of clusters.
3. The ranking selects the most representative graph of each final cluster and reports it to the developer.

We implemented **BigBug** and showed its usability in roughly 2000 experiments. In all of them, **BigBug** reduced the reported violations by orders of magnitude to 6 or less, which equals a median reduction of more than 95%, compared to state-of-the-art concurrency analyzers. Further, we used the output of **BigBug** to fix the bugs in the Floodlight Load Balancer module to show a real-world use case. The evaluation of the fixed application showed the vast majority ($> 99\%$) of all concurrency violations disappeared, which demonstrates the usability of **BigBug** even more. Our evaluation also showed that the trace length does not impact the output of **BigBug**, making long simulations and calculations on huge traces needles. In fact, short traces completely sufficed to find violations in all tested applications. The time needed to execute **BigBug** is rather short for the vast majority of all experiments, making it practical for real-world use.

We believe that **BigBug** is of great use to a developer and greatly reduces the effort needed to find concurrency violations in an SDN application, even though there is still room for improvement as the outlook showed.

Bibliography

- [1] Open Networking Foundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.
- [2] OpenFlow Switch Specification. Version 1.0.0. <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>.
- [3] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: Concurrency Analysis for Software-defined Networks. PLDI '16, 2016.
- [4] Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. SDNRacer: Detecting Concurrency Violations in Software-defined Networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, in ACM SOSR '15.
- [5] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting blackbox sdn control software with minimal causal sequences. *SIGCOMM Comput. Commun. Rev.*, 44(4):395–406, August 2014.
- [6] Floodlight Open SDN Controller. <http://projectfloodlight.org/floodlight>.
- [7] Big Switch Networks, Inc. Floodlight Load-Balancer Application. <https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/loadbalancer>, 2013.
- [8] David Gugelmann, Fabian Gasser, Bernhard Ager, and Vincent Lenders. Hviz: HTTP(S) traffic aggregation and visualization for network forensics. *Digital Investigation*, 12, Supplement 1:S1 – S11, 2015. {DFRWS} 2015 EuropeProceedings of the Second Annual {DFRWS} Europe.
- [9] Tukaram Muske and Alexander Serebrenik. Survey of Approaches for Handling Static Analysis Alarms. In *Proceedings of 16th International Working Conference on Source Code Analysis and Manipulation*, New York, NY, USA, 2016. IEEE.
- [10] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 332–348, New York, NY, USA, 2015. ACM.
- [11] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In ACM PLDI '09.

-
- [12] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 299–314. Springer, 2012.
- [13] Wei Le and Mary Lou Soffa. Path-based fault correlations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 307–316, New York, NY, USA, 2010. ACM.
- [14] T. B. Muske, A. Baid, and T. Sanas. Review efforts reduction by partitioning of static analysis warnings. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 106–115, Sept 2013.
- [15] T. Muske. Improving review of clustered-code analysis warnings. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 569–572, Sept 2014.
- [16] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM.
- [17] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.
- [18] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [19] SciPy Hierarchical Clustering. <https://docs.scipy.org/doc/scipy-0.18.1/reference/cluster.hierarchy.html>.
- [20] Leonard Kaufman and Peter J. Rousseeuw. Clustering by means of medoids. In *Statistical Data Analysis Based on the L1-Norm and Related Methods*, pages 405–416, 1987.
- [21] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.
- [22] Alan Gibbons. *Algorithmic graph theory*. Cambridge University Press, 1985.
- [23] James Mccauley. POX: A Python-based OpenFlow Controller. <https://github.com/noxrepo/pox>.
- [24] Big Switch Networks, Inc. Floodlight Firewall. <https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/firewall>, 2013.
- [25] Big Switch Networks, Inc. Floodlight Circuit Pusher Application. <https://github.com/floodlight/floodlight/tree/v0.91/apps/circuitpusher>, 2013.

- [26] Big Switch Networks, Inc. Floodlight Forwarding Application. <https://github.com/floodlight/floodlight/blob/v0.91/src/main/java/net/floodlightcontroller/forwarding/Forwarding.java>, 2013.
- [27] James McCauley. POX EEL L2 Learning Switch. https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_learning.py, 2015.
- [28] Big Switch Networks, Inc. Floodlight Learning Switch. <https://github.com/floodlight/floodlight/tree/v0.91/src/main/java/net/floodlightcontroller/learningswitch>, 2013.
- [29] James McCauley. POX EEL Forwarding Application. https://github.com/noxrepo/pox/blob/eel/pox/forwarding/l2_multi.py, 2015.

A. Complete evaluation table

App	Topology	Controller	SDNRacer			BigBug			Clusters		Timing		
			Steps	Events	Violations	Isomorphic	Clusters	Timeouts	Final Clusters	Median	Max	Total	SDNRacer
LearningSwitch	BinTree	Floodlight	200	6658	344	210 (61.05 %)	0 (0.00 %)	5 (1.45 %)	48	155	19.075 s	13.753 s	5.323 s
			400	16744	897	450 (50.17 %)	0 (0.00 %)	5 (0.56 %)	108	359	135.686 s	96.995 s	38.691 s
			600	25960	1341	623 (46.46 %)	0 (0.00 %)	5 (0.37 %)	175	535	355.050 s	286.455 s	68.595 s
			800	34236	1874	740 (39.49 %)	0 (0.00 %)	5 (0.27 %)	244	748	591.360 s	488.984 s	102.377 s
			1000	42976	2470	957 (38.74 %)	0 (0.00 %)	5 (0.20 %)	307	961	1303.925 s	1126.017 s	177.908 s
		POX EEL	200	3408	66	61 (92.42 %)	0 (0.00 %)	2 (3.03 %)	33	46	9.819 s	7.208 s	2.611 s
			400	7396	137	103 (75.18 %)	0 (0.00 %)	2 (1.46 %)	50.5	88	45.896 s	28.689 s	17.207 s
			600	11285	175	137 (78.29 %)	0 (0.00 %)	2 (1.14 %)	78.5	135	78.074 s	60.054 s	18.019 s
			800	14625	235	178 (75.74 %)	0 (0.00 %)	2 (0.85 %)	108	186	128.402 s	94.802 s	33.601 s
			1000	17883	260	184 (70.77 %)	0 (0.00 %)	2 (0.77 %)	118.5	195	179.758 s	150.449 s	29.309 s
	Linear	Floodlight	200	228	15	12 (80.00 %)	0 (0.00 %)	2 (13.33 %)	12	14	0.190 s	0.119 s	0.072 s
			400	452	15	10 (66.67 %)	0 (0.00 %)	1 (6.67 %)	8.5	12	3.880 s	3.546 s	0.334 s
			600	717	18	10 (55.56 %)	0 (0.00 %)	2 (11.11 %)	8	13	3.947 s	3.752 s	0.195 s
			800	984	27	17 (62.96 %)	0 (0.00 %)	2 (7.41 %)	13	17	4.357 s	4.032 s	0.324 s
			1000	1248	47	29 (61.70 %)	0 (0.00 %)	3 (6.38 %)	10	21	5.815 s	5.554 s	0.261 s
		POX EEL	200	241	13	6 (46.15 %)	0 (0.00 %)	2 (15.38 %)	6.5	7	0.187 s	0.140 s	0.047 s
			400	490	17	7 (41.18 %)	0 (0.00 %)	2 (11.76 %)	9	14	3.832 s	3.620 s	0.213 s
			600	758	16	7 (43.75 %)	0 (0.00 %)	2 (12.50 %)	13	16	3.989 s	3.896 s	0.093 s
			800	1057	32	12 (37.50 %)	0 (0.00 %)	2 (6.25 %)	20	29	5.421 s	4.845 s	0.576 s
			1000	1287	33	13 (39.39 %)	0 (0.00 %)	2 (6.06 %)	18	31	5.830 s	5.302 s	0.528 s
	Single	Floodlight	200	450	30	10 (33.33 %)	0 (0.00 %)	4 (13.33 %)	7	14	0.402 s	0.334 s	0.069 s
			400	1016	73	19 (26.03 %)	0 (0.00 %)	4 (5.48 %)	18.5	28	5.730 s	4.806 s	0.924 s
			600	1574	126	20 (15.87 %)	0 (0.00 %)	4 (3.17 %)	28.5	41	7.535 s	6.340 s	1.195 s
			800	2185	158	25 (15.82 %)	0 (0.00 %)	4 (2.53 %)	37.5	48	10.347 s	8.721 s	1.626 s
1000			2733	208	28 (13.46 %)	0 (0.00 %)	4 (1.92 %)	55	72	15.214 s	13.479 s	1.735 s	
POX EEL		200	372	6	3 (50.00 %)	0 (0.00 %)	1 (16.67 %)	6	6	0.371 s	0.355 s	0.016 s	
		400	776	16	3 (18.75 %)	0 (0.00 %)	1 (6.25 %)	16	16	4.896 s	4.577 s	0.319 s	
		600	1210	16	4 (25.00 %)	0 (0.00 %)	1 (6.25 %)	16	16	5.453 s	5.228 s	0.224 s	
		800	1620	28	5 (17.86 %)	0 (0.00 %)	1 (3.57 %)	28	28	7.512 s	7.127 s	0.385 s	
		1000	2008	34	5 (14.71 %)	0 (0.00 %)	1 (2.94 %)	34	34	7.621 s	6.988 s	0.633 s	
LoadBalancer	BinTree	Floodlight	200	17593	1910	272 (14.24 %)	0 (0.00 %)	5 (0.26 %)	204	1362	326.107 s	116.644 s	209.463 s
			400	49486	5474	662 (12.09 %)	73 (0.81 %)	5 (0.09 %)	245	4199	4246.278 s	781.849 s	3464.429 s
			600	48895	3611	480 (13.29 %)	9 (0.17 %)	4 (0.11 %)	294	2550	1867.922 s	717.664 s	1150.258 s
			800	45394	2591	419 (16.17 %)	11 (0.34 %)	4 (0.15 %)	371	1649	909.077 s	424.274 s	484.804 s
			1000	68817	5477	608 (11.10 %)	40 (0.48 %)	5 (0.09 %)	311	4089	4911.909 s	1356.859 s	3555.050 s
		Floodlight Fx	200	7626	206	68 (33.01 %)	11 (7.48 %)	3 (1.46 %)	24	177	168.426 s	16.569 s	151.857 s
			400	10863	206	70 (33.98 %)	12 (7.89 %)	3 (1.46 %)	24	179	229.735 s	31.611 s	198.124 s
			600	14845	205	71 (34.63 %)	17 (11.56 %)	3 (1.46 %)	24	175	312.969 s	52.922 s	260.047 s
			800	17905	200	62 (31.00 %)	12 (7.79 %)	3 (1.50 %)	31	169	269.124 s	70.056 s	199.068 s
			1000	21060	205	67 (32.68 %)	12 (7.89 %)	2 (0.98 %)	85.5	184	255.034 s	89.061 s	165.973 s
	Linear	Floodlight	200	2039	225	34 (15.11 %)	0 (0.00 %)	4 (1.78 %)	35	172	6.053 s	4.169 s	1.884 s
			400	5932	861	60 (6.97 %)	0 (0.00 %)	4 (0.46 %)	72	735	76.724 s	36.901 s	39.824 s
			600	9323	1283	74 (5.77 %)	0 (0.00 %)	4 (0.31 %)	116.5	1061	180.790 s	87.350 s	93.440 s
			800	13692	1828	77 (4.21 %)	0 (0.00 %)	4 (0.22 %)	144	1448	280.655 s	173.331 s	107.324 s
			1000	16837	2252	80 (3.55 %)	0 (0.00 %)	4 (0.18 %)	178.5	1908	386.232 s	251.562 s	134.670 s
		Floodlight Fx	200	1341	19	6 (31.58 %)	0 (0.00 %)	3 (15.79 %)	7	10	1.019 s	0.839 s	0.180 s
			400	3387	14	4 (28.57 %)	0 (0.00 %)	2 (14.29 %)	7	10	7.262 s	6.616 s	0.646 s
			600	5262	24	7 (29.79 %)	0 (0.00 %)	3 (12.77 %)	7	11	10.202 s	9.364 s	0.839 s
			800	7197	23	8 (34.78 %)	0 (0.00 %)	3 (13.04 %)	7	12	13.404 s	12.863 s	0.540 s
			1000	9290	23	8 (34.78 %)	0 (0.00 %)	3 (13.04 %)	7	12	17.326 s	16.504 s	0.822 s
	Single	Floodlight	200	4034	1664	107 (6.43 %)	0 (0.00 %)	3 (0.18 %)	125	1529	146.332 s	29.504 s	116.828 s
			400	11445	3281	253 (7.71 %)	0 (0.00 %)	3 (0.09 %)	228	3053	550.670 s	226.691 s	323.978 s
			600	21465	3577	280 (7.83 %)	0 (0.00 %)	3 (0.08 %)	239	3328	1042.517 s	634.503 s	408.013 s
			800	23731	4161	281 (6.75 %)	0 (0.00 %)	3 (0.07 %)	253	3956	1162.911 s	754.363 s	408.548 s
1000			28600	6196	358 (5.78 %)	0 (0.00 %)	3 (0.05 %)	339	5882	1895.051 s	1148.232 s	746.819 s	
Floodlight Fx		200	1385	13	2 (15.38 %)	0 (0.00 %)	2 (15.38 %)	7	8	1.391 s	1.334 s	0.057 s	
		400	3731	13	2 (15.38 %)	0 (0.00 %)	2 (15.38 %)	7	8	9.510 s	9.197 s	0.314 s	
		600	6063	12	2 (16.67 %)	0 (0.00 %)	2 (16.67 %)	7	9	15.172 s	14.748 s	0.424 s	
		800	8591	12	2 (16.67 %)	0 (0.00 %)	2 (16.67 %)	7	9	22.420 s	21.831 s	0.589 s	
		1000	10475	14	2 (14.29 %)	0 (0.00 %)	2 (14.29 %)	7.5	9	28.946 s	28.354 s	0.592 s	