BigBug: Practical Concurrency Analysis for SDN

Roman May, Ahmed El-Hassany, Laurent Vanbever, Martin Vechev
ETH Zürich

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 true violations, limiting their effectiveness in practice.

This paper 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 cause share common characteristics. BigBug leverages these observations to cluster reported violations according to the similarity of events in them as well as 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 100 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.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

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

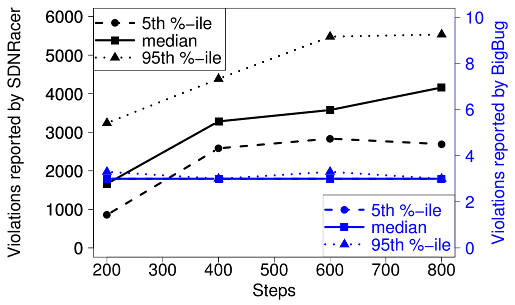

Figure 1: Even when considering one application (Floodlight Load Balancer) running in a star topology with 4 hosts, the number of reported true concurrency violations is huge. BigBug reduces it to 3 representative violations which, when fixed, solves 99.28% of the violations.

1. INTRODUCTION

SDN controllers operate in highly asynchronous environments where events (*e.g.,* packets arriving at a switch) 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 (*i.e.,* switch forwarding table) can often lead to unwanted behaviors and bugs. A classic example is an SDN controller which modifies the 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 [10, 20], a dynamic concurrency analyzer, showed that it is possible to identify true-positive concurrency violations —the real 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. Leveraging HB concurrency analyzers is useful to troubleshoot violations without the need to reproduce them [10, 20].

Problem While a precise concurrency analyzer is a useful first step, often there are too many (thousands) concurrency violations even for short traces. These violations are *not* false positives, and thus a sound HB concurrency analyzer will report them. As an illustration, Fig. 1 depicts the number of violations reported by SDNRacer as a function of the trace length collected on Floodlight Load Balancer application [12, 8]. We can see that an 800 steps trace (~2 minutes) generates no less than *4,000 distinct* concurrency violations! Clearly, trying to sift through such a high number of (actual) violations is practically challenging. In practice, we expect the number of violations to be much higher as SDN controllers tend to run more than one application.

Intuitively, 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 Fig. 1, the controller only contains two distinct bugs. While different violations resulting from the same cause will differ in some ways (*e.g.,* because different packets trigger them), one can expect that they share a common structure.

This work The key idea of our work is an approach and an offline 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 any SDN concurrency analyzer and clusters these reports into equivalence classes. BigBug then selects the most representative violation per class and presents it to the 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 Fig. 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.

Challenges & Solutions 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.

Contributions Our main contributions are:

- A set of domain-specific features to measure the similarities between concurrency violations (§4.1 and §4.2).
- A novel technique to cluster related concurrency violations using the set of domain-specific features (§4).
- A set of ranking techniques which allows BigBug to select a representative candidate of each cluster (§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 100 experiments. In a case study, we also show that solving the bug behind the reported violations caused 99.23% of them to disappear (§6).

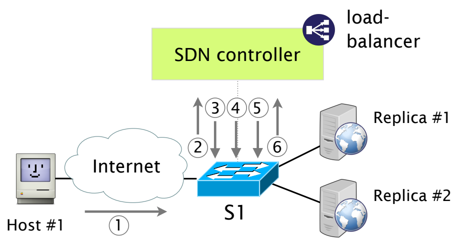

Figure 2: An events sequence creating concurrency violations in Floodlight Load Balancer.

2. OVERVIEW

In this section, we provide a high-level overview of BigBug. We start with a motivating example (§2.1), illustrating how several concurrency violations can result from the same bug in the SDN controller. We then highlight how BigBug (§2.2) clusters a large number 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 (Fig. 2). We assume that an external host, say Host#1, sends a Web request, we call this a Host Send event, which hits S1 ①. As it is a new request, S1 directs it to the controller using an OpenFlow PACKET_IN message ②. The controller selects the best replica, say R1, and sends three OpenFlow messages to S1. The first one is a FLOW_MOD to install a new forwarding entry to forward packets from H1 to R1 ③. The second message is also a FLOW_MOD that forwards packets from R1 to H1 ④. The third message is a PACKET_OUT that carries the original packet sent to the controller ⑤. S1 relies on its flow table to forward that packet.

According to the specification [1], 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 ⑥. This "ping-pong" effect lasts until S1 installs the flow entry 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 or forwarding loops [10, 20].

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 four reasons. *First*, using classical debugging tools that

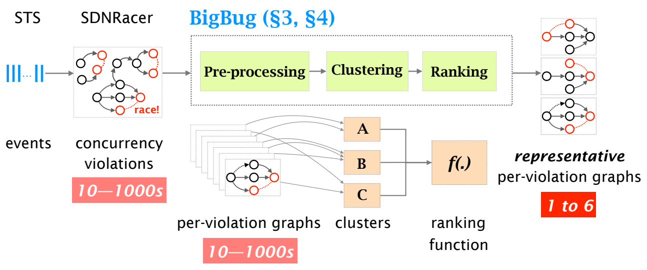

Figure 3: The pipeline of BigBug. Out of potentially thousands of violations, BigBug reduces them to a handful of representative ones which closely map to actual controller bugs.

require replaying the log traces and fixing issues one-by-one is infeasible as the concurrency violations are often non-deterministic and hard to reproduce in test environments. *Second*, 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. *Third*, the number of violations induced by each bug can vary significantly (§6) and does not necessarily correlate with the importance of the problem. For instance, ~90% of the violations reported in Fig. 1 originate from this benign bug. *Fourth*, a developer has no information on the number of bugs that are causing the violations.

2.2 BigBug

To aid the debugging process, BigBug aims to present the developer with only the representative violations which, ideally, correspond 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. 3).

Step 1: Pre-processing Out of a given execution trace, a concurrency analyzer will typically build a directed graph according to a Happen-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 write.

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 §2.1). 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 with 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 experiments), each of the clusters contains many violations, sometimes on the order of 1,000s. In the final step, 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 selects the smallest of these, thus, showing the simplest representative graph.

3. PRE-PROCESSING

BigBug starts by pre-processing the output of SDN concurrency analyzer: 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 §3.1, we show how BigBug reduces the size of the HB-graph. Then, in §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. Such events are filtered by BigBug to reduce the computational complexity of the following stages. Note, BigBug does not detect new violations and it does not remove any from the HB-graph.

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 violation such as the handshake messages between each switch and the controller. *Second*, it removes all events that did not lead to a concurrency violation. *Third*, it removes any redundant HB edges in the HB-graph. about event ordering.

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 the violation graphs vary in size and single event might appear in multiple graphs; if it causes more than violation.

4. HIERARCHICAL CLUSTERING

In this section, we describe BigBug hierarchical clustering process. BigBug first relies on graph isomorphism to initialize the set of clusters (§4.1). BigBug then refines those by grouping related (but not equivalent) violations according to the SDN-specific features (§4.2) they share. For this, BigBug relies on a distance metric (§4.3). We describe the full clustering algorithm in §4.4.

4.1 Cluster initialization

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

In BigBug, we restrict the notion of event equivalence to 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 H with the same type and the same set of edges.

While checking for graph isormorphism can be done in quasipolynomial time [2], it can still take a long time to complete in practice. Therefore we added a timeout of 10 sec for the isomorphism computation. If the timeout is hit, two graphs are considered as not isomorphic and put in different clusters. Observe that they can still be clustered together in later stages A heuristic isomorphic test can be used instead to speed up BigBug cluster initialization (at the cost of accuracy) [9].

4.2 Identifying related violations through SDN-specific Features

As a second step, BigBug uses SDN domain-specific features computed over each graph to compute a distance matrix between clusters. This distance matrix is then used to refine the initial clustering, clustering together closely related (but not equivalent) violations. We identified these features by manually inspecting the similarities among violation graphs of many known bugs traces. Then, we tested the learned features against different known bugs in real controllers. Other features can be discovered using machine-learning techniques.

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.

Formally, let G_k be the set of graphs in cluster C_k, $F_i : G_k \rightarrow \mathbb{N}$ 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 features.

We now present the seven different features currently implemented in BigBug (adding additional ones is easy).

1. **Controller/Switch bouncing:** 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.
2. **Reply packets:** This boolean feature captures if the violation was triggered by a host replying to a packet that it received. Often, the controller simultaneously installs bidirectional path for a flow. The intuition behind this feature is to consider concurrency violations affecting the same flow closer to each other.
3. **Flow expiry:** OpenFlow allows flow entries to expire after a certain specified (hard or soft) timeout [1]. While soft timeout helps cleaning the flow table, defining the timeout is usually tricky in asynchronous environments. Often, early flow expiry lead 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 non-deterministic (hence, not isomorphic). 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 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 one or more events.
6. **Number of host sends:** This feature returns the number of host send events.
7. **Number of proactive violations:** 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 are sent independently. This feature returns the number of proactive events involved in the violation.

Our experiments (§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 *reply packets* one. We capture this effect in the distance function (§4.3) by assigning different weights to each feature.

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

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

Our distance calculation algorithm then computes the distance between every two clusters per-feature. The computation treats boolean and numerical features differently. For boolean features, two clusters are closer to each other if they contain a similar number of occurrences of the feature. For numerical features, two clusters are closer to each other if they share the same mean. Specifically, BigBug computes the per-feature distance between two clusters C_l and C_k as:

$$d_i = abs(m_i^l - m_i^k)$$

if i is a boolean feature

$$d_i = \begin{cases} 0 & \text{if } m_i^l = m_i^k \\ 1 & \text{if } m_i^l \neq m_i^k \end{cases}$$

if i is a numerical feature

We assign different weights for each of the j features (§4.2). The distance between two clusters C_l and C_k is:

$$d = \sum_{i=1}^{j} w_i d_i$$

BigBug computes the distance matrix between all the clusters and then feed it to the clustering algorithm.

When the hierarchical clustering algorithm (§4) groups several clusters into new clusters. For the distance between these groups, we use the distance between the farthest neighbors (also known as *complete linkage*) as the distance between the clusters.

4.4 Clustering Algorithm

We now describe BigBug hierarchical clustering algorithm, which is a case of agglomerative clustering [13].

The six major steps of the algorithm are: *Step 1*, initialize the clusters using the isomorphic check (§4.1). *Step 2*, evaluate all the pair-wise distances (§4.3). *Step 3*, construct a distance matrix using distances values. *Step 4*, merge the cluster pairs with shortest distances and remove them from the distance matrix. *Step 5*, evaluate all distances from this new cluster to all other clusters, and update the matrix. *Step 6*, repeat until the distance matrix is reduced to a single element or the distances are longer than a predefined threshold.

5. RANKING

While the clustering algorithm groups the concurrency violations into a small number of clusters, the number of violations per cluster is large, potentially in the order of 1,000s. 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. For numerical features, 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: 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.

6. EVALUATION

We implemented a working prototype of BigBug and used it to filter out the output produced by SDNRacer [10]. We evaluate on multiple different applications of different controllers on a variety of topologies to show its usability. The results show, that BigBug is of great use in finding representative violations, that hold enough information to fix them and when fixed, reduce 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 traces.

Experimental setup We use BigBug to find violations produced by actual SDN controllers in multiple network topologies (15 repetitions, 200 steps[1]) on a server with two Intel Xeon E5-2670 CPUs and 128GB of RAM. We report the number of violations for Floodlight v0.91 [12] and POX EEL [17] on the following applications:

- Admission Control (Floodlight [5]): Enforces rules to either allow or block communication between hosts.
- Circuit Pusher (Floodlight [4]): Proactively installs routes between any two hosts in the network.
- Forwarding (Floodlight [6], POX [19]): Builds and updates network wide MAC address table.
- Learning Switch (Floodlight [7], POX [18]): Builds and updates MAC address table on per switch basis.
- Load Balancer (Floodlight [8]): Balances the load on multiple hosts behind a virtual IP address.

We generated the traces on three different network topologies: *single*; *linear*; and *binary tree*. The single topology consists of one switch connected to two hosts (we used four hosts for the Load Balancer). The linear

[1]We present results for longer traces at http://sdnracer.ethz.ch/long.pdf. BigBug completes processing traces of 1,000 in less than 1.2 hours in the worst case

		SDNRacer		BigBug		Cluster Sizes	
App	Controller	Events	Violations	Isomorphic	Final Clusters	Median	Max
Adm. Ctrl.	Floodlight	908	81	26 (32.10 %)	**3 (3.70 %)**	24	33
CircuitPusher	Floodlight	1017	39	6 (15.38 %)	**2 (5.13 %)**	19.5	32
Forwarding	Floodlight	3016	288	58 (20.14 %)	**3 (1.04 %)**	31	215
	POX EEL	5632	310	160 (51.61 %)	**4 (1.29 %)**	64.5	143
LearningSwitch	Floodlight	6658	344	210 (61.05 %)	**5 (1.45 %)**	48	155
	POX EEL	3408	66	61 (92.42 %)	**2 (3.03 %)**	33	46
LoadBalancer	Floodlight	17593	1910	272 (14.24 %)	**5 (0.26 %)**	204	1362

Table 1: BigBug performance on traces computed over a binary tree topology (200 steps, median on 15 repetitions).

topology consists of two hosts that are connected via two switches. The binary tree consists of seven switches with four host connected to the leaves.

A simple sensitivity analysis led us to use the following weights for the distance function (§4.3): Controller/Switch Bouncing, Packet Flood, and Flow Expiry have weight 2. Number of Proactive Violations has weight 1.5. Number of Host Sends has weight 1. Number of root events and Reply Packets have weight 0.5. The maximum distance for the merging of clusters was set to 2. We are aware that different weights can result in an even better (or worse) clustering and leave a full sensitivity analysis for later work.

Usability BigBug reports a small number of concurrency violations to the developer, moreover, fixing only these reported violations signficatly reduces the number of violations exhibited by the controller.

Table 1 shows that BigBug is able to reduce the number of reported violations by up to three orders of magnitude. Clustering isomorphic graphs already reduces the number of violations by more than 66% in 50% of all cases. The feature-based agglomerative clustering further reduces them to less than 95% of the reported violations of SDNRacer in 50% of the cases.

To demonstrate the usability of BigBug, we used the reported violations by BigBug to fix the Floodlight Load Balancer application. When tested on the fixed version, the number of clusters reported by BigBug drop dropped from 3 to 2 and the number of violations was reduced by 99.23%.

It is worth mentioning that not all concurrency violations in SDN networks are fixable. This due the inherent lack of OpenFlow synchronization primitives that order packets upon entering or while traversing the network. However, recent SDN systems give the controller the ability to synchronize packet entry to the network [25].

Performance For the experiments reported in Table 1, BigBug finished in less than one second for 60 % of all the experiments, and only 5% of the experiments took more than 216 seconds with a worst case of 21 minutes (for Floodlight Load Balancer on binary tree).

The 10 seconds timeout for isomorphic check was triggered in only 2.08% of the checks of POX EEL Forwarding Application on binary tree. This application has a flooding-related concurrency violation which creates large violation graphs spanning the entire network.

BigBug can process subsets of the trace (windowing) which would be helpful to troubleshoot longer traces. The intuition here is that individual races are usually "concentrated" and do not last over the entire trace.

7. RELATED WORK

Grouping or clustering concurrency violation reports has been applied for event-driven concurrency analyzers in other domains [3, 16, 23, 21, 15, 22]. The main difference is that 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 [3]). Also, BigBug does not rely on static analysis and actually considers the controller code as a black box. Thanks to this, our clustering approach based on HB information is general and can thus benefit existing analyzers such as [3].

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 [11, 14, 24].

8. CONCLUSION

In this paper we introduced BigBug, a generic framework to automatically narrow down the most representative concurrency violations, i.e. the ones that better illustrate the likely root cause of an actual bug. To do so, BigBug clusters the violations reported by concurrency analyzers into few equivalence classes (using graph isomorphism and SDN-specific features) before reporting the most relevant violation in each class.

We implemented BigBug and show that it is practically effective. In all our experiments, BigBug reduced the reported violations to 6 or less. More importantly, fixing the bugs behind the violations reported by BigBug made the vast majority of the violations disappear.

9. REFERENCES

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

[2] L. Babai. Graph Isomorphism in Quasipolynomial Time. *Computing Research Repository (CoRR)*, abs/1512.03547, 2015.

[3] P. Bielik, V. Raychev, and M. Vechev. Scalable Race Detection for Android Applications. In *OOPSLA*. ACM, 2015.

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

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

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

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

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

[9] D. G. Corneil and D. G. Kirkpatrick. A Theoretical Analysis of Various Heuristics for the Graph Isomorphism Problem. *SIAM Journal on Computing*, 1980.

[10] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev. SDNRacer: Concurrency Analysis for Software-defined Networks. In *PLDI*. ACM, 2016.

[11] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In ACM PLDI '09.

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

[13] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[14] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS XVII*. ACM, 2012.

[15] W. Le and M. L. Soffa. Path-based Fault Correlations. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010.

[16] W. Lee, W. Lee, and K. Yi. Sound Non-Statistical Clustering of Static Analysis Alarms. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012.

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

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

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

[20] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. SDNRacer: Detecting Concurrency Violations in Software-defined Networks. In *SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM, 2015.

[21] T. Muske. Improving Review of Clustered-Code Analysis Warnings. In *Software Maintenance and Evolution (ICSME)*. IEEE, 2014.

[22] T. Muske and A. Serebrenik. Survey of Approaches for Handling Static Analysis Alarms. In *Proceedings of 16th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2016.

[23] T. B. Muske, A. Baid, and T. Sanas. Review Efforts Reduction by Partitioning of Static Analysis Warnings. In *Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013.

[24] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*. ACM, 2007.

[25] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*. ACM, 2014.