Poster: A Benchmark for Event-Race Analysis in Android Apps

Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
{nsalehna,aalshayb,iftekha,malek}@uci.edu

ABSTRACT

Over the past few years, researchers have proposed various program analysis tools for automated detection of event-race conditions in Android. However, to this date, it is not clear how these tools compare to one another, as they have been evaluated on arbitrary, disjointed set of Android apps, for which there is no ground truth, i.e., verified set of event races. To fill this gap and support future research in this area, we introduce *BenchERoid*, a set of 34 Android apps with injected event-race bugs. The current version of benchmark contains 36 types of event-race bugs that were identified by analyzing Android concurrency literature and publicly available issue repositories. We believe that our framework is a valuable resource for both developers and researchers interested in concurrency bug analysis in Android. *BenchERoid* is publicly available at: https://github.com/seal-hub/bencheroid.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Event-Race, Benchmark, Android, Concurrency

ACM Reference Format:
Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Poster: A Benchmark for Event-Race Analysis in Android Apps. In *The 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '20), June 15–19, 2020, Toronto, ON, Canada.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3386901.3396602

1 INTRODUCTION

Concurrency mechanisms in Android are an important source of confusion for developers and contribute to many software bugs in Android apps [6]. The Android platform provides a hybrid concurrency mechanism supporting both traditional multi-threaded and event-driven programming paradigms [2]. An app defines many *callback methods* to handle *events* originating from the user (e.g., tap on the screen), the system (e.g., notifications of location change), other apps (e.g., requesting data), or the app itself (e.g., downloading a file in a background thread). In addition, Android provides a variety of concurrency libraries for use by developers, e.g., `Looper` or `AsynTask`.

MobiSys '20, June 15–19, 2020, Toronto, ON, Canada

ACM ISBN 978-1-4503-7954-0/20/06.
https://doi.org/10.1145/3386901.3396602

Among the various concurrency bugs, *event race* is not only the most frequently encountered, but also the most challenging to detect in Android apps [3, 5, 6]. An event race occurs when two event handlers access the same memory location, where at least one of the accesses is a write operation, and there exists no specific order between their executions; therefore, the final value contained in the memory location can change depending upon the stochastic execution order of events [3].

Over the past few years, researchers have proposed various program analysis tools for automated detection of event-race conditions in Android [3, 5, 6]. Despite these efforts, not much is known as to their relative strengths and weaknesses. This is largely due to the fact that the evaluation of these tools have been performed in an ad hoc fashion. The subject apps used for the evaluation of these tools are neither available (the source code and bug locations are not provided) nor representative (the authors randomly chose a handful of apps without specifying any selection criterion). Research in this area is hindered by the lack of a benchmark for which the ground truth is known.

Exiting concurrency benchmarks for Java [4, 8] are not suitable for Android due to the differences between the platforms. For instance, in Android, event races can occur even within a single thread [3], since the events may execute without any specific order, while in Java event races can only occur in scenarios involving multiple threads. Figure 1 shows an event-race example in Android. When the `Activity` is created, it registers a `BroadcastReceiver` (line 6). The `BroadcastReceiver` updates a widget, `textView`, when it receives a message (line 4). The widget is initialized in `onStart` and assigned a `null` value when the activity goes to the background in `onStop` (lines 7-8). Here, if an intent is sent to the receiver when the app is in the background, a *null pointer exception* is thrown, forcing a crash. This event race is caused by the non-deterministic execution order between `onReceive` and `onStop` even though both are running on the main thread.

While the research community has developed popular benchmarks for various quality concerns, such as security vulnerabilities [1] and energy bugs [7], surprisingly none exists for event-race issues in Android. To fill this gap, we introduce *BenchERoid*, a first of its kind, publicly available dataset of Android apps containing a variety of event races (https://github.com/seal-hub/bencheroid). We meticulously reviewed real world apps and recent publications to identify entities that contribute to the manifestation of event-race bugs in Android apps. Next, we handcrafted 34 Android apps harboring examples of such concurrency bugs involving a variety of entities. Finally, we built and ran all the apps, and validated the existence of event races. The validated event races provide the ground truth for evaluating the accuracy of event-race detection tools and advancing the research in this area.

```
1 class MainActivity extends Activity {
2  TextView textView;
3  BroadcastReceiver receiver = new BroadcastReceiver() {
4   public void onReceive(...) { textView.setText("
       Received!"); }
5  };
6  void onCreate(){ registerReceiver(receiver, new
       IntentFilter(...)); }
7  void onStart(){ textView = findViewById(...); }
8  void onStop(){ textView = null;} }
```

Figure 1: An event-race example in Android

2 METHODOLOGY

In order to make *BenchERoid* comprehensive, we followed a systematic approach of identifying the various entities that may contribute to the occurrence of event-race conditions in apps. We started by collecting the relevant publications from well-known program analysis research venues such as PLDI, OOPSLA, FSE, ASE, CGO, ASPLOS, MobiSys, and ICSE for the past 10 years starting from 2010. Our initial search identified 46 research papers, out of which we filtered the ones that are relevant to our study (8 papers related to event races in Android). We also searched for Github issues and StackOverflow questions with a set of keywords related to event races such as "Event race", "Concurrency bugs", "Order Violation", etc. Our search resulted in 4,173 issues in Github and 177 StackOverflow questions. The first two authors independently read the papers, reviewed the issues and questions, and sifted through the corresponding apps to identify the entities that affect the occurrence of event-race conditions in Android. Next, the authors discussed their findings and arrived at 9 entities, as discussed further below.

- **Concurrency API.** An event race is caused by concurrent invocations of instructions handled by a concurrency library, e.g., `AsyncTask`.
- **Synchronization Mechanism.** The concurrency behavior of an app can be managed using synchronization mechanism, e.g., `wait` in Thread.
- **Involved Components.** Android components, e.g., Activity, are created and managed asynchronously by the framework.
- **Android Callbacks.** An Android app may override some predefined Android callbacks which can be *Life Cycle* callbacks of components, e.g., `onCreate`, or *Event* callbacks, e.g., `onClick`.
- **Inter-Component Communication.** Different components can communicate with each other concurrently by sending intent messages.
- **Input.** The execution of events depends on the provided input. For example, an event race may occur only when specific inputs match the condition of an `if`.
- **Execution Order.** The order of execution of statements may cause (or prevent) event races.
- **Thread.** Events are executed on threads with various properties, e.g., a Looper handles its messages sequentially. These properties may introduce or prevent an event race.
- **Time-sensitivity.** The order of execution of events may be dependent on the time of their execution.

For each entity, we identified a set of possible values (Table 1) by analyzing the literature. We created combinations of these values to generate meaningful event-race scenarios. For example, an event race can occur while using `TimerTask` concurrency API that depends on the time of execution. In total, we created 34 apps for 568 pairwise combinations that cover a wide variety of event-race scenarios. We further added some infeasible event races along with true ones for evaluating the precision of event-race detectors. All apps have been developed in Android Studio 3.5 and can be built automatically using Gradle. The detailed list of apps and event races are provided as part of the companion website.

Table 1: Different entities and their possible values along with the number of apps, possible event races and true event races.

Entity	Values	#Apps	#Possible ER	#True ER
Concurrency API	Thread	4	8	5
	Handler	11	25	12
	AsyncTask	3	13	9
	Executor	2	2	2
	Timer	2	2	2
	IntentService	2	2	1
Synchronization Mechanism	Wait/Notify/Join	2	4	2
	Looper	7	19	8
	with Delay	4	8	5
	Front Of Queue	2	4	1
	Serial	5	6	2
	Parallel	4	10	7
Involved Components	Activity	34	60	36
	Service	5	7	5
	BroadcastReceiver	1	3	3
Android Callbacks	Life-Cycle	21	49	27
	Event	14	29	19
ICC	Yes/No	6	10	7
Input	Yes/No	4	9	4
Execution Order		10	24	7
Thread		10	24	12
Time-sensitivity		3	5	2

3 CONCLUSION

In this paper, we presented *BenchERoid*, a micro benchmark for supporting the analysis of event-race conditions in Android apps. The benchmark includes 34 apps containing 36 types of concurrency bugs. We release the benchmark under the Apache License (Version 2.0) and make it publicly available on Github. Our goal is to provide the community with the means to support the research on detection of concurrency bugs in Android and *BenchERoid* was the first step towards fulfilling that goal. Our future work involves comparative analysis of existing tools using *BenchERoid*.

REFERENCES

[1] 2020. *DroidBench.* Retrieved March 24 , 2020 from https://github.com/secure-software-engineering/DroidBench

[2] 2020. *Looper in Android.* Retrieved March 28 , 2020 from https://developer.android.com/reference/android/os/Looper

[3] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable race detection for Android applications. In *2015 International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015.* ACM Press, 332–348.

[4] Yaniv Eytani, Rachel Tzoref, and Shmuel Ur. 2008. Experience with a concurrency bugs benchmark. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop.* IEEE, 379–384.

[5] Xinwei Fu, Dongyoon Lee, and Changhee Jung. 2018. nAdroid: statically detecting ordering violations in Android applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization.* ACM Press, 62–74.

[6] Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-based Races in Android Apps. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 257–270.

[7] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: an energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 208–219.

[8] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. Jacontebe: A benchmark suite of real-world java concurrency bugs (T). In *IEEE/ACM International Conference on Automated Software Engineering.* IEEE, 178–189.