# Systematic Asynchrony Bug Exploration for Android Apps

Burcu Kulahcioglu Ozkan[1(✉)], Michael Emmi[2], and Serdar Tasiran[1]

[1] Koç University, Istanbul, Turkey
{bkulahcioglu,stasiran}@ku.edu.tr
[2] IMDEA Software Institute, Madrid, Spain
michael.emmi@imdea.org

**Abstract.** Smartphone and tablet "apps" are particularly susceptible to asynchrony bugs. In order to maintain responsive user interfaces, events are handled asynchronously. Unexpected schedules of event handlers can result in apparently-random bugs which are notoriously difficult to reproduce, even given the user-event sequences that trigger them.

We develop the AsyncDroid tool for the systematic discovery and reproduction of asynchrony bugs in Android apps. Given an app and a user-event sequence, AsyncDroid systematically executes alternate schedules of the same asynchronous event handlers, according to a programmable schedule enumerator. The input user-event sequence is given either by user interaction, or can be generated by automated UI "monkeys". By exposing and controlling the factors which influence the scheduling order of asynchronous handlers, our programmable enumerators can explicate reproducible schedules harboring bugs. By enumerating all schedules within a limited threshold of reordering, we maximize the likelihood of encountering asynchrony bugs, according to prevailing hypotheses in the literature, and discover several bugs in Android apps found in the wild.

## 1 Introduction

Android apps execute asynchronously: typically a number of background threads exist to prevent long-running tasks from tying up the main UI thread. Threads execute asynchronously-called procedures concurrently with other threads. Programmers tend to imagine atomically-handled events, without taking all possible thread interleavings into consideration. However, event handlers often call other asynchronous methods, and so the execution of multiple events can interleave and result in hard-to-reproduce bugs.

In this work we present AsyncDroid[1], the first concurrency testing tool for Android apps. AsyncDroid takes a sequence of user events given by user interaction or automated UI "monkeys" and explores different thread interleavings to

This work is supported in part by the Scientific and Technological Research Council of Turkey (TUBITAK).

[1] http://github.com/imdea-software/async-droid.

© Springer International Publishing Switzerland 2015
D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 455–461, 2015.
DOI: 10.1007/978-3-319-21690-4_28

detect thrown exceptions and assertion violations. Focusing on the systematic exploration of alternate schedules of asynchronously-executing methods, and prioritizing those schedules derived from few re-orderings, our technique uncovers many violations quickly, and uncovers all violations given enough time.

Our prototype implementation explores all deviations from a base schedule within a user-specified bound. In addition to providing a default thread scheduler from which the base schedule can be generated, we also provide an interface allowing users to implement their own scheduler to guide the exploration process along their own insights. Besides control over the thread schedules, our implementation can also record a given sequence of user events, and then replay the same sequence events along alternate thread schedules. We implement both scheduler control and event recording/replaying via program instrumentation, without modifying Android runtime libraries.

*Related Work.* Existing approaches to bug detection in Android apps fall into two basic categories. The first category focuses on UI input testing [1,2,4,9]. Orthogonally to these techniques, which test a single execution of any given UI-event sequence, our goal is to explore the alternate schedules of execution for a given UI-event sequence, thus uncovering elusive concurrency-related bugs. The second category of techniques investigates race detection [3,7,8,10]. Our work is complementary to these techniques and it is novel in two respects: it is a dynamic analysis rather than static, and it does not report false positives.

While prioritized systematic exploration of concurrent program executions has been studied before [5,11], the adaptation to event driven programs poses some specific difficulties. A tool must explore different possible concurrent behaviors for a given, fixed user interaction with the program. This requires the recording and replay of user input events while exploring alternate schedules. Moreover, it is nontrivial to design effective thread schedulers for the typical Android "looper" threads, which do nothing but execute the handlers of received messages.

## 2 Design and Implementation

The basic functionality of AsyncDroid is to repeat a sequence of UI events over a systematic enumeration of thread schedules. We achieve this functionality via a program instrumentation which provides explicit control over thread scheduling, and recording/replaying of UI events for given thread schedules.

### 2.1 Recording and Repeating User Events

To record the user events, we instrument each visible UI component with an additional event handler. When run in record mode, the instrumented handler records each event and forwards it to the original handler. This allows us to capture both direct user interaction, and simulated "monkey" interaction.

In replay mode, we use an input repeater which reads and replays the recorded events for every thread schedule to be tested. The input repeater runs

in its own thread and feeds the user events to the application concurrently to the execution of the other threads. AsyncDroid schedules this thread as well as the other application threads, controlling the interleaving between sending an event and the execution in the other threads.

In approaches in the literature, the user events are recorded by saving the coordinates of an input event and replayed by giving the same input on the same coordinates [6]. However, the timing of the inputs differ in each schedule and a UI component might not be visible at a time we want to replay it. Repeating an event using only the coordinates might result in the invocation of a wrong event, since a different view might exist on the original event's input coordinates at the time of replay. To overcome this, we use an abstraction of an input event close to the application semantics. Our event abstraction keeps the path to UI component of the user event. While replaying an event, we make sure that the recorded path to an event fully matches to the view on the currently visible UI.

### 2.2 Thread Scheduling

AsyncDroid controls the scheduling of the input repeater, UI, and background threads. To explore different execution schedules, AsyncDroid treats the beginnings and ends of asynchronous methods as scheduling points, only preempting threads at these points to determine a complete schedule.

AsyncDroid's default scheduler runs threads until becoming blocked in a round-robin fashion. The input repeater thread is enabled if it has more events to replay and the next event's UI component is visible. Similarly, the UI thread and other threads are enabled if they have some tasks to execute. In Android, it is likely that the UI thread have repetitive runtime tasks (for interprocess communication, UI update, etc.) in its queue and never becomes disabled during an execution. In this case, the standard preemption bounding approach would spend a preemption to switch from the UI thread. Our tool blocks a thread and switches to another thread when the message queue of a current thread is empty or it has only recurring Android-runtime messages.

Though the default scheduler runs the threads in round-robin fashion, we use delay bounding [5] to prioritize our search of alternate executions. For a given bound $k$, we systematically explore all executions which correspond to thread schedules which are $k$-delay deviations from the default schedule.

AsyncDroid also allows the programmer to specify his own default scheduler by implementing certain scheduling hooks. To this end, we provide an interface which exposes the current list of application threads, whether each thread is blocked, the list of pending events in the input-repeater thread, and the lists of pending tasks in the UI and background threads. The programmer and access these lists in deciding which thread to dispatch at any given scheduling point.

## 3 Case Studies

As an instructive case study, we investigate an asynchrony bug in the Vlille Checker app used with the public bicycle-sharing program in the city of Lille,

France.[2] The app displays a list of bicycle stations together with their status and information. The user can (un)mark a station as a favorite, and limit their view to favorite stations. While the list is being viewed, station information is updated asynchronously in a background thread to keep the UI thread responsive.

The following scenario triggers our bug, depending on the execution order of asynchronous methods. Figure 1 shows the relevant application code with distinguished statements labeled L1, L2, and L3.

– The user clicks to view their favorites list.

L1 To initiate the status update of the favorite stations which are currently visible on the screen, the application creates a sublist of visible favorite stations. Crucially for the bug in question, this sublist is not represented by a new data structure, but is instead backed by the same data structure as the full list of favorite stations.

– The user clicks to remove a station from the favorites list.

L2 An asynchronous task executing on the UI thread removes the station from the favorites list.

L3 An asynchronous task executing on a background thread iterates over the visible favorites list in order to update their statuses.

Since L2 and L3 are executed asynchronously on separate threads, they can execute in any order, depending on hard-to-determine system scheduling factors. In the case that L2 is executed before L3, the ArrayList constructor throws a *ConcurrentModificationException* as the favorites list backing the visible favorites sublist has been modified.

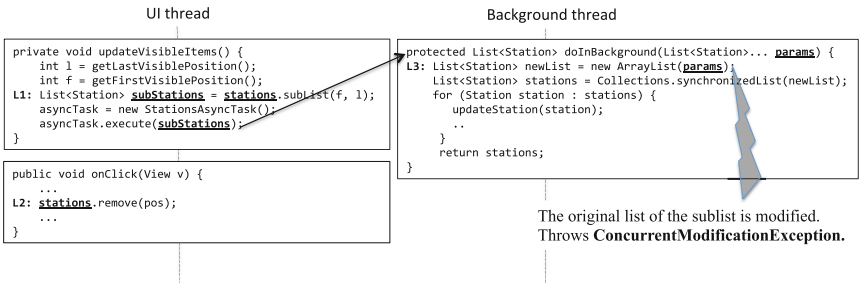


**Fig. 1.** An exception thrown only in executions of the Vlille Checker app in which the list removal at Statement L2 is executed between the sublist creation at L1 and its use in the constructor at L3 of the asynchronously-called doInBackground method.

To produce the bug, we record the following event sequence and systematically explore possible schedules of asynchronous methods:[3]

[2] The bug report: https://github.com/ojacquemart/vlilleChecker/issues/60.

[3] The test which produces this bug is available on AsyncDroid's Github repository.

1. click on a menu item to display all stations,
2. click on a station to add it into the favorites,
3. click on a menu item to display the favorite stations, and
4. click on the favorite station to remove it from the favorites.

Note that many schedules of asynchronous methods for this event sequence do not expose the bug. For instance, without incurring delays, AsyncDroid's default scheduler runs each thread until it is no longer enabled before moving on to the next thread in a round-robin fashion. The input repeater thread becomes blocked after Event 3 is actuated, since Event 4 is not enabled until the favorite stations list becomes visible. Next, the scheduler executes the pending UI-thread tasks, causing the asynchronous *doInBackground* method to become pending on the background thread. Once the UI thread becomes idle, the scheduler executes the background task to completion before returning to the input repeater thread where Event 4 is enabled, the favorite stations list having been updated and made visible. In this way, our default scheduler, without delaying, executes Statement L3 before L2, and does not expose the bug.

However, by enumerating all 1-delay executions, AsyncDroid does discover an execution in which the *ConcurrentModificationException* is thrown, by delaying the background thread before executing Statement L3. This delay causes our default scheduler to return to the input repeater thread where Event 4 is enabled, due to the UI thread having made the favorite stations list visible. After actuating Event 4, we return to the UI thread to process its *onClick* handler before returning to the background thread. This 1 delay execution thus executes Statement L2 before L1, and throws the exception.

We also applied our tool to the ACV comics and image viewer app[4], the Jamendo online music player app[5], and a hand-crafted microbenchmark with an injected asynchrony bug. We test the ACV app by providing tap inputs to browse, view, and rotate images. Since tap inputs are always enabled in this app, independently of the app state, our replay is not guaranteed to be faithful to the original event sequence, e.g., in the case that certain taps are ignored in certain states. This limitation could be overcome by more precise tracking of UI state.

The Jamendo music player app is tested by browsing, selecting and playing a radio channel. While playing music, recurring messages are sent to the UI thread to display track progress. As AsyncDroid repetitively runs all schedules without restarting the app, these recurring tasks remain in the message queue after the execution of a schedule completes. This causes the next schedule to start with some leftover tasks. AsyncDroid overcomes this problem by calling an optional finalizer method implemented by the programmer in his app itself to clean up the tasks in the message queues. We tested the Jamendo app by adding a finalizer method into its source code.

Our hand-crafted microbenchmark inserts into and deletes from a list of items, in response to user events. When the user wants to insert an item, it

[4] https://github.com/robotmedia/droid-comic-viewer.
[5] https://github.com/telecapoland/jamendo-android.

**Table 1.** Quantitative results of our case studies.

| | # of events | # of switch | # of sch. dec | 0 delay | | 1 delay | |
|---|---|---|---|---|---|---|---|
| | | | | # of conf. | Bug? | # of conf. | Bug? |
| Vlille checker | 5 | 9 | 29 | 30 | No | 59 | Yes |
| | 7 | 8 | 33 | 34 | No | 69 | Yes |
| ACV comic viewer | 6 | 9 | 19 | 14 | No | 31 | No |
| | 8 | 9 | 23 | 19 | No | 40 | No |
| Jamendo music player | 3 | 23 | 100 | 69 | No | 87 | No |
| | 5 | 21 | 37 | 24 | No | 61 | No |
| Microbenchmark | 3 | 5 | 7 | 8 | No | 10 | Yes |
| | 5 | 5 | 11 | 12 | No | 19 | Yes |

increases the items count and performs the insertion in a background thread. If the removal of the item in the last index is processed before the background thread, the app throws an *IndexOutOfBoundsException*.

Table 1 lists the quantitative results of our case studies. As a rough measure of behavioral coverage, we measure the number of "abstract" program configurations encountered in each exploration, which distinguish only the number of asynchronous tasks pending on each thread. For each run, the table depicts the length of a fixed input event sequence, the number of context switches and the number of scheduling decision points encountered in the zero-delay execution, and the number of abstract configurations encountered. AsyncDroid quickly reproduces the known, yet previously nondeterministically-occurring, bugs in the Vlille Checker and our microbenchmark in a matter of minutes using a single delay. While we do not know whether the ACV and Jamendo apps contain a bug, AsyncDroid does not discover one within a single delay.

## 4 Limitations and Future Work

Applying systematic concurrency exploration to event driven programs faces the fundamental obstacle that some schedules may become infeasible due to the unavailability of a given UI component at a given time. When the component corresponding to a scheduled-for-replay event is not visible on the screen, we delay its activation, disrupting the intended schedule. This limitation raises research questions about how to integrate the treatment of UI events in concert with systematic concurrency exploration.

Controlling all scheduling decisions is a key implementation challenge. In our current prototype, we focus on the systematic analysis of interleavings between the asynchronous methods created by the given app, and leave the scheduling of other asynchronous methods (e.g., periodic system events) uncontrolled.

AsyncDroid currently only supports recording and replaying of certain types of UI events: we handle simple clicks, but not text inputs nor gestures. Capturing a wider set of UI events will allow us to test a larger set of applications. Our future work also involves developing coverage metrics to evaluate how various scheduling strategies compare with respect to coverage of program behaviors.

## References


1. Anand, S., Naik, M., Harrold, M.J., Yang, H.: Automated concolic testing of smartphone apps. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 59:1–59:11. FSE 2012, ACM, New York, NY, USA (2012). http://doi.acm.org/10.1145/2393596.2393666
2. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 641–660. OOPSLA 2013, ACM, New York, NY, USA (2013). http://doi.acm.org/10.1145/2509136.2509549
3. Bielik, P.: Effective Race Detection for Android. Master's thesis, ETH Zurich, Switzerland (2014)
4. Choi, W., Necula, G., Sen, K.: Guided GUI testing of Android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. pp. 623–640. OOPSLA 2013, ACM, New York, NY, USA (2013). http://doi.acm.org/10.1145/2509136.2509552
5. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 411–422. POPL 2011, ACM, New York, NY, USA (2011). http://doi.acm.org/10.1145/1926385.1926432
6. Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: Reran: Timing- and touch-sensitive record and replay for Android. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 72–81. ICSE 2013, IEEE Press, Piscataway, NJ, USA (2013). http://dl.acm.org/citation.cfm?id=2486788.2486799
7. Hsiao, C.H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C.L., Pokam, G.A., Chen, P.M., Flinn, J.: Race detection for event-driven mobile applications. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 326–336. PLDI 2014, ACM, New York, NY, USA (2014). http://doi.acm.org/10.1145/2594291.2594330
8. Lin, Y., Radoi, C., Dig, D.: Retrofitting concurrency for Android applications through refactoring. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 341–352. FSE 2014, ACM, New York, NY, USA (2014). http://doi.acm.org/10.1145/2635868.2635903
9. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for Android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 224–234. ESEC/FSE 2013, ACM, New York, NY, USA (2013). http://doi.acm.org/10.1145/2491411.2491450
10. Maiya, P., Kanade, A., Majumdar, R.: Race detection for Android applications. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 316–325. PLDI 2014, ACM, New York, NY, USA (2014). http://doi.acm.org/10.1145/2594291.2594311
11. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 446–455. PLDI 2007, ACM, New York, NY, USA (2007). http://doi.acm.org/10.1145/1250734.1250785