Flaky Test Detection in Android via Event Order Exploration

Zhen Dong
National University of Singapore, Singapore
zhen.dong@comp.nus.edu.sg

Abhishek Tiwari
National University of Singapore, Singapore
dcsabhi@nus.edu.sg

Xiao Liang Yu
National University of Singapore, Singapore
xiaoly@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore, Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Validation of Android apps via testing is difficult owing to the presence of flaky tests. Due to non-deterministic execution environments, a sequence of events (a test) may lead to success or failure in unpredictable ways. In this work, we present an approach and tool *FlakeScanner* for detecting flaky tests through exploration of event orders. Our key observation is that for a test in a mobile app, there is a testing framework thread which creates the test events, a main User-Interface (UI) thread processing these events, and there may be several other background threads running asynchronously. For any event e whose execution involves potential non-determinism, we localize the earliest (latest) event after (before) which e must happen. We then efficiently explore the schedules between the upper/lower bound events while grouping events within a single statement, to find whether the test outcome is flaky. We also create a suite of subject programs called *FlakyAppRepo* (containing 33 widely-used Android projects) to study flaky tests in Android apps. Our experiments on the subject-suite *FlakyAppRepo* show *FlakeScanner* detected 45 out of 52 known flaky tests as well as 245 previously unknown flaky tests among 1444 tests.

CCS CONCEPTS

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

KEYWORDS

flaky tests, non-determinism, concurrency, event order

ACM Reference Format:
Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2021. Flaky Test Detection in Android via Event Order Exploration. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3468264.3468584

1 INTRODUCTION

Regression testing aims to discover the adverse effects of the recently added code changes. Ideally, a test failure during regression testing should reliably signal issues with recent code modifications. However, some test failures are not due to the recent updates but due to *flaky tests*. Recent research [6, 17, 30] establishes flaky tests as tests that non-deterministically pass or fail when running on the same code version. Unfortunately, the presence of flaky tests significantly harms developers' productivity [1, 25, 30].

Android is a reactive system, and its non-deterministic execution environment often causes concurrency-related flaky tests. In Android, a GUI test usually simulates user interactions to exercise the app's functionality. However, due to non-determinism, a test could generate such interactions at "incorrect" timings. For example, a test to download an image and then open it without considering the internet's speed could show non-deterministic behavior. A potential reason for such concurrency issues is the lack of synchronization among threads, e.g., in the failing run, a background thread would still be downloading the image while the testing thread tries to open it without synchronization with the background thread.

Recent studies [34, 42] confirm that such synchronization issues lead to a significant number of flaky tests in Android apps. Thorve et al. [42] studied 77 flakiness-related commits in Android projects and discovered that 36% failures are due to synchronization issues between testing thread and app under test, e.g., a test accesses data in the app before the data is available. Similarly, Romano et al.'s study [34] shows that 33% flaky test failures are caused by threads' synchronization issues (tests interacting with the UI elements before the elements are fully loaded).

Challenges. Exposing a concurrency-related flaky test in a test suite is challenging as such flakiness is only observed when events get executed in a particular order. In Android, a test's event execution order may show non-determinism due to the non-deterministic execution environment. Some of such (irregular) event orders may cause a test to fail/pass occasionally. Consequently, detecting such a situation would require exploring such event orders deterministically; unfortunately, this poses a set of challenges for the existing flaky test detection approaches, which we discuss below.

- *RERUN.* A typical way to detect a flaky test is to rerun (expressed as *RERUN*) it multiple times. If it passes in some runs and fails in others, the test is marked as flaky. However, this approach can struggle to detect concurrency-related flaky tests because the target's execution environment may not introduce the needed non-determinism. Besides, it may require too many runs to witness the flakiness.
- *Noise Based.* Several approaches [4, 38, 40, 48] run tests with environmental perturbations (e.g., changing CPU load or test execution orders) in the hope of observing unexpected behaviors.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468584

Despite being simple and easy to be adopted in practice, adding noise does not guarantee to explore different execution orders. Consequently, they fail to detect many concurrency related flaky tests, as observed during our experiments.

- *Event Race.* The concurrency-related flakiness may be caused due to a synchronization problem on the testing thread. Besides, a concurrency bug on the app under test could also cause a non-deterministic test failure. Thus, in principle, approaches identifying concurrency bugs should also detect a subset of such flaky tests. Recent works [8, 18, 19, 24, 32] leverage program analysis techniques to analyze possible races in Android apps. For instance, *EventRacer* [8] records the execution trace of a test and statically analyzes it to find potential races. However, such approaches are prone to multiple false positives, which may create more problems for the developers during the Continuous Integration (CI) process.

Our Approach. We introduce a lightweight technique that automatically detects a concurrency flaky test in Android apps within few test runs. Our technique explores the possible event execution orders in a test by scheduling a non-deterministic (async) event such that each test run explores a different event execution order. Besides, we identify and explore event orders in which the test flakiness manifests such that a flaky test can be detected in a few test runs.

Insight. Possible event execution orders are introduced by non-deterministic execution of *async events*. Android apps use a concurrent event-driven model, in which only the main (UI) thread can access GUI and processes user events. To keep GUI responsive, the UI thread offloads a long-running task (e.g., internet accessing) to a background thread. Once the task is finished, the background thread sends an event (called *async event*) to the UI thread to perform a GUI update. In a test run, the testing thread and background threads send events to the UI thread simultaneously, resulting in races among the events; the test may pass for some event orders and fail in others. For the example mentioned earlier, the testing thread simulates the "open the image" event, and the background thread downloads the image and sends an update event after completing the download. A race may occur between these two events. If the internet is fast, the update event reaches the UI thread before the "open the image" event, and the test passes. Otherwise, the "open the image" event is processed first, and the test fails.

Event Order Exploration. Leveraging these insights, our approach explores possible event execution orders by scheduling an async event in a test run. Our approach first identifies the schedule space for each async event with dynamic analysis and schedules the async event in its schedule space to avoid infeasible orders. To manifest a flaky test failure quickly, our approach schedules events at positions where the test is more likely to fail. Specifically, it schedules an async event at a *statement boundary position*. The statement boundary position is between the last event that a test statement triggers and the first event that the next statement triggers. We note that the tests are often flaky due to missing an appropriate synchronization operation between two test statements, e.g., the synchronizing operation after clicking a download button. Thus we are more likely to trigger a test failure if an async event is executed after executing certain test statements.

Instrumentation-free Tool. We implement our approach into a tool called *FlakeScanner*, which leverages the debug mode supported in the Android framework to perform dynamic analysis based event scheduling. Thus *FlakeScanner* requires no instrumentation in either Android apps or the Android framework and works on both Android emulators and devices. According to the study [41], code instrumentation often disrupts test executions and prevents the manifestation of flaky test failures. Moreover, *FlakeScanner* supports multiple widely-used testing frameworks with which developers write tests such as Espresso [2] or Robotium [13].

Experiment. We evaluated *FlakeScanner* on 33 widely-used Android projects. Our experiments show that *FlakeScanner* successfully detected 45 out of 52 known flaky tests. On average, it detected a flaky test within three test runs. *FlakeScanner* outperformed the recently published flaky test detection technique *Shaker* [40] and the baseline tool RERUN in terms of both the number of detected flaky tests and average execution time. *FlakeScanner* also detected 245 flaky tests that were previously unknown. Out of these 245 unknown flaky tests, we reported 20 to developers; 13 out of these 20 have been confirmed and addressed by developers.

Contributions. Our contributions can be summarized as follows.

- We present an event scheduling approach that explores different event execution orders for each test run to detect concurrency-related flaky tests in Android apps. To avoid exploring all possible event orders, our approach adopts heuristics to identify and explore event orders in which the test flakiness is likely to occur.
- We implement our approach into an instrumentation-free tool that works on Android emulators and physical devices while supporting widely-used Android testing frameworks.
- We curate a suite of subject programs containing 33 widely-used Android projects with developer tests, called *FlakyAppRepo* for evaluating flaky test detection techniques. To facilitate future research on flaky tests, we make our prototype *FlakeScanner* and subject-suite *FlakyAppRepo* available at https://github.com/AndroidFlakyTest

2 BACKGROUND

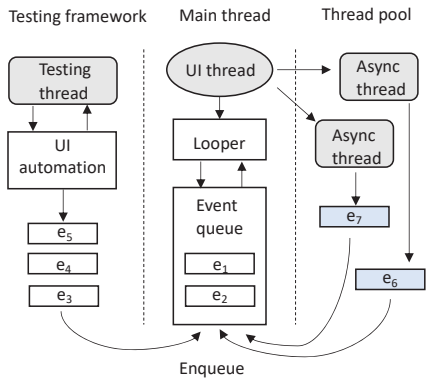

Figure 1: Android concurrency model & testing framework.

Android Concurrency Model. Figure 1 depicts Android's event-driven concurrency model. Each Android app has a main thread (also called *UI thread*); this thread processes events generated by users or the Android system. As shown in Figure 1, the UI thread maintains an event queue and an event looper. The queue is used to store received events, and the looper sequentially dequeues events from the queue and dispatches them to corresponding handlers for processing. Android adopts a single-UI-thread model in which only the main thread can access GUI objects. To achieve rapid UI responsiveness, the UI thread offloads long-running tasks such as network access to background threads, called *async threads* since they communicate with the UI thread asynchronously. Once tasks are completed, async threads post an event (called *async event* marked in blue in Figure 1) to the UI thread, and the UI thread updates the results to GUI. Event races may occur in this model. The UI thread and async threads run concurrently. The duration that an async thread will take to complete a task and post an async event is non-deterministic, depending on the current execution environment. Thus, a race might occur among posted async events and others, leading to non-deterministic execution orders. In the example in Figure 1, there are multiple event orders which might occur in the execution, for instance, $\langle e_1, e_2, e_6, e_3, e_4, e_5, e_7 \rangle$ and $\langle e_1, e_2, e_3, e_4, e_5, e_6, e_7 \rangle$.

Instrumented Tests & Testing Frameworks. Instrumented tests are tests that run on physical devices and emulators, and they can invoke the Android framework APIs to control app under test at runtime. These tests are often executed in a separate thread (called *testing thread*) to simulate user interactions, as shown in Figure 1. The Android system supports multiple testing frameworks to help developers write tests, e.g., Espresso [2]. To achieve reliable tests, these frameworks provide mechanisms to synchronize user interactions with app under test. For instance, when method `onView()` is invoked in a test, Espresso waits to perform the corresponding UI action or assertion until the event queue is empty, background threads are terminated, and user-defined resources are idle.

3 A MOTIVATING EXAMPLE

To set the stage for our event order exploration technique for flaky test detection, we first illustrate the characteristics of a flaky test and then exemplify current approaches' inadequacy through Listings 1-3. These listings show parts of *CaptureLocationActivityTest*, a flaky test taken from *RapidPro Surveyor* app. This test intends to validate whether the app can successfully obtain the location data using Google APIs. As shown in Listing 1, the test launches *CaptureLocationActivity* (Line 2), the activity to capture the location data (Listing 2). Later, the test emulates a button click to fetch the location at line 5 in Listing 1, and validates whether the location is successfully fetched at line 7 in Listing 1.

Despite appearing straightforward, the test displays non-deterministic behavior, i.e., it is flaky. As explained earlier, the test runs in a testing thread, and the activity under test runs in the app's UI thread. The activity (*CaptureLocationActivity*) offloads fetching location data via Google API client to an async thread (Listing 3). After obtaining the location data, the async thread updates the result to the UI thread, and then the UI thread updates this result to *CaptureLocationActivity*. Due to the lack of synchronization between the testing thread and async thread, the async thread might send the location data before or after the testing thread validates it. If the validation occurs before the location data is received, the test fails, else it passes.

Detecting concurrency-related flaky tests is non-trivial as such failures manifest when the events are executed in a specific order. The traditional approach *RERUN* blindly executes the test many times in the hope of witnessing the flaky test failures. However, this approach becomes ineffective when the environment under which the test runs does not produce the required non-determinism. For example, *RERUN* failed to witness the flaky test failure in *CaptureLocationActivityTest* during our experiments. Another approach is to add noise in the execution environment to increase the likelihood of observing such errors. However, such approaches do not proactively explore different event execution orders and are prone to miss crucial event orderings. In our evaluations, the recent (noise-based) related work *Shaker* [40] failed to detect many confirmed flaky tests, including *CaptureLocationActivityTest*. We also evaluated *EventRacer*, a dynamic analysis based technique for detecting event races in Android apps. However, such approaches are prone to report many races. For example, in our evaluation, *EventRacer* reported over 200 data races for *CaptureLocationActivityTest*. Unfortunately, none of these reported data races from *EventRacer* involve events originated from the testing thread, and hence do not demonstrate the specific flakiness we are illustrating in this section. Besides, *EventRacer* does not validate whether reported races can cause the test failure.

4 OVERVIEW OF OUR APPROACH

This section describes the main components of our framework and lists the principles that make event exploration suitable for detecting concurrency-related flaky tests. First, the category of flaky tests should be well defined. It is acceptable to give up on all classes of flaky tests and focus on one; concurrency-related flaky tests. Second, using only one test run, we explore (all) possible scheduling space of an async event. Third, we execute these new schedulings to observe the flaky behavior, i.e., there exist at least two executions for which a test depicts different outcome (pass/fail).

4.1 Basic Concepts

Before diving into details of our approach, we define a miniature domain-specific syntax extended from [20] for an Android test.

Test	T	::=	$\overrightarrow{S}$
Statement	S	::=	$post(\overrightarrow{e})$ | ASSERTIONS | SYNC | *other*
Event	e	::=	$\overrightarrow{M}_{UIThread}$ | $\overrightarrow{M}_{backgroundThread}$ | $\overrightarrow{M}_{OS}$
Synchronize	SYNC	::=	SYNC($\overrightarrow{S}$)
Message	M		

A test T is composed of a series of program statements S. For concurrency-related flaky tests, we are interested in exploring the scheduling space of events, and thus for ease of representation, we categorize a test statement into four parts; statements posting an event, assertion statements, testing framework-specific synchronizing statements, and all other Android-specific statements. An event, denoted as e, is a message object created in the UI thread

Listing 1: CaptureLocationActivityTest (Testing Thread)

```
1 @Rule
2 public ActivityTestRule<CaptureLocationActivity> rule = new
      ActivityTestRule<>(CaptureLocationActivity.class);
3 @Test
4 public void capture() {
5   onView(withId(R.id.button_capture))
      .check(matches(isDisplayed())).perform(click());
6   Instrumentation.ActivityResult result =
      rule.getActivityResult();
7   assertThat(result.getResultData(), is(not(nullValue())));
8   ...
9 }
```

Listing 2: CaptureLocationActivity (UI Thread)

```
1 @Override
2 protected void onCreate(Bundle bundle) {
3   connectGoogleApi();
4   ...
5 }
6 protected void connectGoogleApi() {
7   googleApiClient = new GoogleApiClient.Builder(this) ...
8   googleApiClient.connect();
9 }
```

Listing 3: Zaau (Worker Thread)

```
1 //Worker thread asynchronous to the UI thread
2 @WorkerThread
3 public void run() {
4   zaak.zac(this.zagj).lock();
5   ...
6 }
```

($\overrightarrow{M}_{UIThread}$), background threads ($\overrightarrow{M}_{backgroundThread}$), or Android framework ($\overrightarrow{M}_{OS}$). Each event specifies a specific action to perform, e.g., a button click registers an *on-click* event. Synchronizing statements (SYNC) are testing framework-specific methods and are used to achieve synchronization among UI Thread and testing thread, e.g., Espresso uses `onView()` method to achieve synchronization among testing thread and UI Thread[1]. Based on above constructs, we define an async event e_{async} as the event generated from a background thread, i.e, $e_{async} \in \overrightarrow{M}_{backgroundThread}$.

Definition 4.1 (Scheduling Space of Events). Let T be a test containing a set of statements $\overrightarrow{S}$, $\overrightarrow{e}$ be the sequence of events generated by $\overrightarrow{S}$. Let $\overrightarrow{e}$ contains n async events and $\overrightarrow{e}'$ be a new sequence of events created by reordering an async event in $\overrightarrow{e}$. Then, the scheduling space of events $\overrightarrow{\mathcal{E}}$ is a set of all possible $\overrightarrow{e}'$.

Given a test, Definition 4.1 formally defines scheduling spaces of events. Intuitively, in a test, the order of an async event is not fixed, and thus, creating new event orders by reordering async events will provide all potential event orders. However, these event orders may also contain (infeasible) orders that will not be realized in practice. One of the critical challenges is to avoid such orders. An infeasible event ordering can be explored by ignoring the events' dependencies during the execution. For instance, an async event e_1 may depend on another event e_2, where event e_1 will not complete before e_2 in practice. Based on this observation, we define infeasible event orders as:

Definition 4.2 (Infeasible Scheduling Space). Let $\overrightarrow{e}$ be a sequence of events and $\overrightarrow{e}'$ be a new sequence created by reordering an async event e_{async} in $\overrightarrow{e}$ from the position i to a new position j. Then, $\overrightarrow{e}'$ will be infeasible if $\exists e_k \in \overrightarrow{e}'$ after the position j and e_{async} depends on e_k.

Intuitively, *depends* specifies the *Happens-before* relation. In practice, avoiding infeasible event orders would require computing the dependencies among events in the test execution. However, computing event dependencies is challenging for multiple reasons. First, a test can trigger many events (more than 500 events for some cases). Computing dependencies among them can be complex as one event's execution may depend on multiple other events. Second, event dependencies are not specified in the events but handed over to the components that respond to the events. These components may belong to Android frameworks or third-party libraries, making event dependency analysis difficult. We propose a dynamic analysis to explore various event execution orders to address this challenge. Given an async event e in a test run, the analysis identifies the scheduling space $\langle e_{lo}, e_{up} \rangle$, where e_{lo} is *lower bound event* and e_{up} is *upper bound event*, i.e., e cannot be executed earlier than e_{lo} or later than e_{up} for all execution environments. Consequently, we can explore event execution orders by scheduling the async event e at various positions between e_{lo} and e_{up}.

4.2 Identifying the Scheduling Space

The successful realization of the scheduling space requires noticing two crucial points. First, the scheduling space needs to reflect some level of determinism, i.e., there need to be some non-async events. Second, each async event's scheduling space should be well defined.

Observation 1. *For a statement s in a test T, the first event it generates will not be an async event, i.e., its order will not change for all runs.*

Android is a reactive system, and tests in an Android app invokes the functionality of Android's components (e.g., an activity) by simulating user events. For example, a test would emulate a button click to invoke some functionality provided by an activity. The concept of background threads in Android is trivial, and Android's components utilize them to offload long-running tasks in the background. Notably, a background thread is always created by an Android component (via UI Thread). Thus, the interaction from a test (via a statement) to an Android component would always follow a sequence where the test would invoke the component, and then the component may invoke a background thread. Considering such event chaining, the first event of a test statement will always belong to $\overrightarrow{M}_{UIThread}$ or $\overrightarrow{M}_{OS}$. Based on this observation, we define *anchor events* as:

Definition 4.3 (Anchor Events). Let $s \in \overrightarrow{S}$ be a program statement in a test T, and it generates n events in the following order: $< e_1, ..., e_n >$. Then, we define e_1 as the anchor event for s.

To localize anchor events, we execute a test, statement by statement; we record all events triggered by each statement, and build

[1] https://developer.android.com/training/testing/espresso#sync

a map between them. According to this map, we identify anchor events for the test. Defining the lower bound of an async event is now straightforward. Intuitively, for a given statement, the lower bound of an async event would always be the anchor event of this statement, as the anchor event will always *happen-before* async event (Observation 1).

Localizing the upper bound event of an async event e_{async} is more involved and requires identifying events that depend on e_{async}, i.e., events that occur only after e_{async}. Such event dependence is maintained by thread synchronization operations. For instance, the testing framework Espresso uses `onView()` operation to synchronize the testing thread and app under test. The testing thread waits until specific threads or resources are idle to ensure that a widget specified by `onView()` shows up on the screen. Formally, we define the upper bound of an async event as:

Definition 4.4 (Upper Bound Event). Let T be a test containing a set of statements $\overrightarrow{S}$. We define e_j as the upper bound event for an async event e_{async}, where e_j is the first event that cannot *happen-before* e_{async}.

We propose a *what-if* analysis to localize the upper bound event of e_i. Specifically, after the test is launched, we hook the async thread that posts e_i at runtime and suspend it, keeping other threads free. Meanwhile, we monitor the testing thread to check at which statement the testing thread stops and waits for the suspended thread to be completed. Suppose that the testing thread stops at a statement s on executing the event e_j. We then deem operations in s to depend on e_i, and these operations will not be executed until e_i is processed. Thus, the anchor event e_j triggered by s is upper bound event of e_i. The idea behind the analysis is *what-if* it takes forever to complete the long-running task that corresponds e_i; operations in a test that depend on e_i will not be executed due to thread synchronization, and those that do not depend on e_i will be executed.

4.3 Scheduling Events

Next, we explore the scheduling strategies for an async event e_{async}. Given the lower and upper bound events of e_{async}, the anchor events that lie between this interval are identified. Then, e_{async} is scheduled before each of these anchor events. To manifest flaky test failures as soon as possible, we prioritize positions closer to the upper bound event (maximizing the runtime of the async event), i.e., we first schedule e_{async} just before its upper bound and then move towards its lower bound.

For two reasons, we only explore positions before anchor events in the scheduling space of an async event instead of all possible positions. First, for non-anchor events in the scheduling space, their execution orders may change from run to run. Using them as hooks for event scheduling may introduce infeasible orders. Second, anchor events are boundary events of test statements. Scheduling an async event before them is likely to trigger a flaky test failure.

5 METHODOLOGY

Figure 2 shows the workflow of our approach. For a given test, our framework performs a concrete execution to trace all the generated events. Next, a map between the statements in the test and their

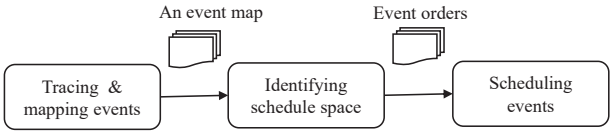

Figure 2: Workflow of exposing flaky tests through systematic schedule exploration.

corresponding events is created. The test is executed multiple times to compute possible schedules for async events. Consequently, a set of event orders that might occur in execution environments are created. Finally, we explore these possible event execution orders; if a test failure is detected during the exploration, the test is identified as a flaky test (since we have already seen passing runs of the test).

5.1 Event Tracing and Mapping

Event tracing is often used in the dynamic analysis of Android apps. It can be achieved by simply logging events that are generated at runtime. However, such techniques cannot fulfill our task. Event information (*e.g.*, event id) produced in logs is dynamically generated by Android runtime and changes in each run. Our approach requires an event identifier to identify an event uniquely across different test runs. Async events that are identified during event tracing need to be hooked and scheduled in runs that are performed for event order exploration.

Event Identification. We identify an event based on interactions between the event and app under test at runtime. Two events triggered in different test runs are considered identical if: (1) they are triggered by the same test statement; (2) they are processed by the same sequence of methods at runtime. For instance, a `pressDown` event is associated with an identifier constructed using the line number of the statement that triggers the event and signatures of a sequence of methods that process it. This practice of event identification comes from our investigation of the Android framework.

Tracing and Mapping. Algorithm 1 outlines the procedure of event tracing and mapping. First, it launches the app under test and takes control of the Android runtime with a module called *ARTHandler*. Given a test T consisting of a series of statements, ARTHandler runs the test T in the testing thread and executes statements one by one (via the Android debugger). When one statement is executed, ARTHandlers monitors the event queue of the UI thread and hooks injected events. For each event, ARTHandler records the tuple $\langle isAsync, sq_m \rangle$ where *isAsync* denotes whether it is an async event and sq_m denotes the signatures of a sequence of methods that have processed the event. This tuple and the line number of the statement being executed form the event's identifier and get stored in a list (Line 11-14). As stated before, a statement in the test might perform long-running tasks, which are executed in async threads. When these tasks are completed is non-deterministic, and the async event might be posted after a long time. To not miss async events that are triggered by a single statement, we keep hooking events until two criteria are satisfied: (a) there are no new events and (b) the event queue of the UI thread is empty, which often indicates the system is not running tasks. This practice is also used in the

Algorithm 1: Event tracing and mapping.

```
1  Procedure runTest(App A, Test T, Android ART)
2      launchApp (A, ART);
3      ARTHandler ← attachHandler (A, ART);
4      List ← ∅ ; // storing pairs of a statement and events
5      launchTest (A, T, ART);
       // UI thread's event queue
6      eventQ ← getEventQ(ARTHandler);
7      for s in T do
8          runStatement(ARTHandler, s);
9          n ← getLineNum(ARTHandler, s);
10         while True do
11             E⟨isAsync, sq_m⟩ ← getEvent(ARTHandler);
12             if E != Null then
13                 List ← List ∪ {⟨isAsync, sq_m, n⟩}
14             else
15                 if isEmpty(eventQ) then
16                     break ;
17                 end
18             end
19         end
20     end
21     return List
```

Algorithm 2: Event scheduling exploration.

```
1  Procedure explore(App A, Test T, Android ART, EventMap List)
2      for e in List do
3          if isAsync (e) then
4              launchApp (A, ART);
5              ARTHandler ← attachHandler (A, ART);
6              launchTest (A, T, ARTHandler);
7              TH_test ← getTestingThread (ARTHandler);
8              TH_async ← hookAsyncThread (ARTHandler);
9              suspend (ARTHandler, TH_async) ;
10             while True do
11                 if isWaiting (TH_test) then
12                     break ;
13                 end
14             end
15             m ← getLineNum (ARTHandler, TH_test);
16         end
           // m is set as upper bound for e
17         setUpperBound (e, m) ;
           // update event e (with upper bound m) in List
18         updateEvent (List, e) ;
19         break ;
20     end
21     return List
```

Espresso testing framework. Finally, a map between a statement and its events is returned via the *List*, and the first event of each statement is identified as an anchor event.

5.2 Identifying Event Schedule Space

To compute possible event execution orders, we perform a *what-if* dynamic analysis to resolve event dependencies caused by thread synchronization between apps and testing frameworks. Algorithm 2 shows the procedure of resolving event dependencies. It takes the event trace *List* generated in the previous step as input. For each async event e_{async} in *List*, the algorithm launches the test and starts to hook event e_{async} (Line 4-8). Once hooked, the algorithm suspends the thread that posts e_{async} such that e_{async} cannot be posted (Line 9). Meanwhile, it keeps checking the status of the testing thread (Line 10-14). If the testing thread's status is `WAITING`, it considers the testing thread is performing thread synchronization with threads in the app and waiting for e_{async} to be executed. Thus, we consider the statement s that is being executed in the testing thread attempts to trigger an event (e.g., e_j) which depends on e_{async}. Therefore, the schedule space of e_{async} is bounded by e_j, i.e., the first event that is triggered by s. So statement s is identified as the upper bound of schedule space of async event e_{async}. Statement s is recorded and set as the upper bound of event e_{async} and e_{async} is restored to *List*. Finally, schedule spaces for all async events in *List* are recorded.

5.3 Scheduling Events

Schedule space of each async event in the event trace *List* is identified in the previous steps. Next, we explore event orders during test execution. An async event e_{async} can be simply represented by a triplet $\langle id, n, m\rangle$, where n and m are bounds of the scheduling space of e_{async}. Specifically, n is the index of the statement in the test that triggers e_{async}, and m is the index of the statement that triggers the upper bound event of e_{async}.

Similar to scheduling space identification, we can schedule e_{async} by operating threads. We first hook event e_{async} after the test is launched and suspend the thread that posts e_{async}. Then, we free the testing thread and monitor whether the statement being executed is statement m. Once statement m is reached, we suspend the testing thread and free the suspended thread to post e_{async}. After the async thread is terminated or idle, i.e., event e_{async} has been posted, we free the testing thread. In such a way, event e_{async} can be executed before statement m. In next test run, we schedule e_{async} to be executed prior to statement $m-1$, until all statements between n and m are explored. This procedure is repeated for each async event in *List*.

6 IMPLEMENTATION

Our system is implemented in Scala and runs on a computer that connects a physical Android device or an emulator. Unlike existing techniques, it requires no instrumentation on apps or the Android framework and can be adapted to different versions of Android.

Taking Control of Android Runtime. The Android framework supports running an app in the debug mode, under which the Android runtime can be fully controlled. Specifically, we connect the Android runtime via Android Debug Bridge (ADB) and use the Android debugger to execute the app under test. With the debugger's help, we can perform execution step by step and monitor the app state during thread manipulation.

Hooking Events. Android adopts the event-driven model and manages events using an event queue. Each event implements an interface method called `enqueueMessage()` and Android puts

Table 1: Subject Apps

App Name	Version	#LOC	#Stars	Category
Amaze File Manager	3.4.3	92.2k	2.8k	Tools
Youtube Extractor	2.0.0	2.7k	519	Video Players
AntennaPod	1.8.1	102.6k	2.7k	Music & Audio
Backpack Design	2.0.7	84.2k	363	Productivity
Barista	3.5.0	67.9k	1.2k	Productivity
CameraView	2.6.1	40.5k	2.9k	Photography
Catroid	0.9.69	457.5k	0	Education
City Localizer	1.1	4k	0	Travel & Local
ConnectBot	1.9.6	119.7k	1.4k	Communication
DuckDuckGo	5.43.0	211.3k	1.2k	Tools
Espresso	1.0	17.3k	1.1k	Maps & Navigation
Firefox Focus	8.5.1	44.5k	1.6k	Communication
Firefox Lite	2.18	1598.4k	212	Communication
FlexBox	2.0.1	29.2k	15.2k	Libraries & Demo
GnuCash	2.4.0	90.1k	1k	Finance
IBS FoodAnalyzer	1.2	26.1k	0	Health & Fitness
Google I/O	7.0.14	73.5k	19.6k	Books & Reference
Just Weather	1.0.0	5.9k	65	Weather
Kaspresso	1.1.0	66.3k	774	Productivity
KeePassDroid	2.5.3	159.7k	1.2k	Tools
KickMaterial	1.0	79.1k	1.6k	Crowdfunding
KISS Launcher	3.11.9	27.2k	1.4k	Personalisation
MedLog	1.0	65k	0	Medical
Minimal To Do	1.2	27.5k	1.8k	Productivity
MoneyManagerEx	02.14.994	170k	265	Finance
My Expenses	3.0.7.1	1835.6k	248	Finance
NYBus	1.0	6.9k	272	Transport
Omni Notes	6.0.5	105.9k	2k	Productivity
OpenTasks	1.2.4	448k	724	Productivity
ownCloud	2.14.2	333.7k	2.9k	Productivity
Sunflower	0.1.6	5.3k	10.1k	Gardening
Surveyor	13.3.0	290.4k	13	Communication
WordPress	14.2-rc-2	461.7k	2.3k	Productivity

events to the queue by calling this method. We set a breakpoint at method `enqueueMessage()` in the debug mode. Whenever an event is generated and injected into the queue, we perform predefined operations such as suspending the event-posting thread.

Operating Threads. We leverage the Android debugger to retrieve threads running in the app under test, including the testing thread and UI thread. The Android debugger provides commands to remotely operate threads, e.g., inspecting thread status and suspending a selected thread. It also allows us to obtain the current stack frames of a running thread that are used for event identification. With the Android debugger, we can execute a testing thread step by step and observe execution at the statement level by inserting a breakpoint at each statement in the test.

7 EVALUATION

We empirically evaluate *FlakeScanner*'s effectiveness in detecting flaky tests in the test suites of large-scale Android projects.

7.1 Subject Apps

Our technique is designed for a test that runs on the Android framework platform and checks whether it is flaky. To evaluate our technique, we need Android projects that contain such tests. Basically, there are two types of tests in Android projects: *instrumented tests* that run on a physical device or Android emulator and *local unit tests* that run on local Java virtual machines. Thus, we need Android projects that contain instrumented tests. Unfortunately, most Android projects in existing benchmarks such as AndroTest [11], industrial app benchmark [44], and data set that is used for flaky test empirical study [42] have no instrumented tests.

Table 2: Root causes of reproduced flaky test failures and their categories.

Categories	Description of root causes
Async	Do not wait or wait not enough when accessing background resources or services in an async manner for tasks such as image rendering, downloading from internet, and invoking third party libraries.
Event orders	Expecting an implicit event execution order that may not always occur in the test execution. The unexpected event execution order leads to app misbehavior such as the soft keyboard disappearing late.
Data race	Checking unsynchronized data because checking occurs before the data being updated due to the lack of thread synchronization.
Lifecycle	Performing app-state-sensitive operations on the incorrect state, e.g., the testing thread attempts to operate GUI widgets when the app is resumed state, which is prohibited.

Therefore, we built the first subject-suite *FlakyAppRepo* in which each Android project contains instrumented tests that are written by developers. As shown in Table 1, *FlakyAppRepo* contains 33 Android projects (majority of them are well-known, such as WordPress) and over 5000 instrumented tests from developers. We collected Android projects containing instrumented tests as follows: (1) searching popular open-source Android projects such as Firefox and manually checking their repositories to select projects with instrumented tests (intuitively, popular projects are well maintained and more likely have developer tests); (2) searching keywords such as *flak*, *flakiness*, or *intermit* on the Github and manually checking searched repositories to select Android projects containing instrumented tests. Furthermore, we explored these keywords in the commit history as well to include already fixed flaky tests. A few searched projects have 0 star on the Github. To achieve diversity, we did not exclude these projects and kept them in the subject-suite.

7.2 Research Questions

Our evaluation aims to address the following research questions.

RQ1 Can *FlakeScanner* detect known *flaky tests* that are reported by developers?

RQ2 How does *FlakeScanner* compare with existing techniques in terms of number of detected flaky tests?

RQ3 Can *FlakeScanner* be used to discover flaky tests in Android projects that were previously unknown?

7.3 Experiment Setup

To answer the research questions, we conduct three empirical studies on test cases that are written by developers in real world Android app projects.

7.3.1 *Study 1.*

We first evaluate *FlakeScanner*'s effectiveness in flaky test detection by running it on known flaky tests in Android projects and checking how many of them are marked as flaky tests by *FlakeScanner*.

Table 3: Dynamic analysis based tools for detecting flaky tests or concurrency issues in Android apps.

Approach	Categories	Instrumentation	Year	Reason for not selected
RERUN	Flaky tests	No	-	✓
Shaker [40]	Flaky tests	No	2020	✓
APEChecker [16]	Async bugs	No	2018	publicly unavailable
ERVA [20]	Event race	Yes	2016	Publicly unavailable
EventRacer [8]	Event race	Yes	2015	✓
AsyncDroid [24]	Async bugs	Yes	2015	Incompatible instrument API

Data Set. We selected 52 known flaky tests from *FlakyAppRepo* that are caused by concurrency or synchronization issues for Study 1. In Android projects, flaky tests reported by developers are marked with a dedicated annotation `@FlakyTest` such that flaky tests can be automatically filtered out during test execution. We identified 269 known flaky tests in *FlakyAppRepo* using `@FlakyTest` annotation. We selected known tests from them in the following manner.

Concurrency We select concurrency related flaky tests by manually analyzing (1) details about reported flaky tests such as the reason of why a test is flaky, which can be collected from `detail` element [2] of `@FlakyTest` annotation; (2) commit messages when flaky tests were introduced or fixed in a subsequent version; (3) whether there exist statements in tests that are commonly involved in concurrency or synchronization execution, e.g., operations of `Runnable` or `AsyncTask` objects.

Passing We ran each selected test in our execution environment and ensured it passes in the initial run, avoiding failures due to environment setup.

Reproducing We reproduce the flaky test failure for each selected test by analyzing its commits messages and detailed description in the repository including root causes and failing scenarios. For cases lacking the description, we contacted developers to seek more insights on the failure reproduction. A test is selected if its failures is reproduced.

In the end, 52 known flaky tests were selected for the evaluation. Meanwhile, we also analyzed root causes of these flaky test failures and classified them into four categories which are shown in Table 2.

7.3.2 *Study 2.*

To answer RQ2, we evaluate *FlakeScanner* and existing tools on the data set used in Study 1. We reviewed most recent works on flaky tests or event race detection for Android apps and summarize them in Table 3. We selected the following approaches for comparison.

- *RERUN.* In practice, developers often run a test many times to check whether the test is flaky. Similarly, we run a test 100 times and check whether the flaky test failure manifests during execution. We call the approach *100RUN* and take it as our baseline.
- *Shaker* [40] is the most recently reported state-of-the-art technique for detecting concurrency-related flaky tests in Android apps. *Shaker* attempts to manifest a flaky test failure by adding noises in the execution environment to affect event execution order, e.g., changing CPU workload.
- *EventRacer* [8] is a the-state-of-art dynamic technique of detecting event races in Android apps, which infers races by analyzing runtime traces produced by an instrumented Android framework.

7.3.3 *Study 3.*

To answer RQ3, we run *FlakeScanner* on tests in *FlakyAppRepo* without `@FlakyTest` annotation, i.e., tests that are not reported as flaky tests. We first run these tests in our execution environment and exclude tests that fail, then feed passing tests to *FlakeScanner* and report tests that are labeled as flaky tests by *FlakeScanner*.

Effect of debug mode. FlakeScanner runs apps in the debug mode. The debug mode environment may affect flaky test detection by impacting the Android event dispatching mechanism or increasing execution workload. According to the official Android document[3], the difference between normal execution mode and debug mode is that the Android virtual machine (VM) loads an additional Android Runtime Tooling Interface (ARTTI) plugin that exposes runtime internals (e.g., variable values). The ARTTI plugin runs as a VM-level component and it is transparent to event dispatching that occurs in the app's main thread, i.e., running apps in debug mode does not impact the mechanism of event dispatching.

However, running the ARTTI plugin and accessing runtime internals in the execution could increase the workload of the VM. Due to the workload change, a test can manifest the flaky test failure. To rule out such cases, for each test in our study, we pre-execute it to ensure the test passes in the debug mode environment. To perform end-to-end comparison in Study 2, we run *100RUN* and *Shaker* in the debug mode environment as well.

Execution Environment. We conducted experiments on a physical machine with 64 GB RAM and a 56 cores Intel(R) Xeon(R) E5-2660 v4 CPU, running a 64-bit Ubuntu 16.04 operating system. Each execution instance runs in a Docker container to minimize the potential inference between running instances. App under test runs on an Android 9 (x86) emulator. One execution instance is for one test case for which the Android emulator is initialized to a fresh state at the beginning to provide a clean testing environment.

7.4 RQ1: Efficacy

Table 4 shows results on each known flaky test for study 1. The first column indicates known test Ids, the second column shows app names and testing frameworks used in apps, and the third column indicates test method names. Column "#Event" indicates the number of events observed by *FlakeScanner* during detection. "#Run" indicates the number of test runs in the event order exploration. Column "Time" reports the time that is used to detect a flaky test. Column "Succ" indicates whether the test is identified as a flaky test by *FlakeScanner*. "-" indicates that a test is unable to be executed due to library compatibility issues. Firefox-Lite marked with "*" (rows 39-40) is a previous version (commit:465739510e). Note that some test names in the table are shortened for readability.

Results. FlakeScanner successfully detected 45 out of 52 known flaky tests that are from 10 Android projects (including a different version of Firefox Lite). On average, *FlakeScanner* detected a flaky test in 101 seconds. The maximum detection time is for test 20 from app AntennaPod (691 seconds). The minimum detection time is for

[2] https://developer.android.com/reference/androidx/test/filters/FlakyTest

[3] https://source.android.com/devices/tech/dalvik/art-ti

Table 4: Results on known flaky tests by *FlakeScanner*, *Shaker* and *100RUN*.

Id	App: Framework	Method name	*FlakeScanner*				Shaker		100Run	
			#Events	#Run	Time(s)	Succ	Time(s)	Succ	Time(s)	Succ
1	Surveyor: Espresso	capture	58	2	48	✓	4301.76		80	
2		twoQuestions	104	2	48	✓	2639.1	✓	1236	
3		multimedia	233	8	542	✓	1607.28		486	
4		contactDetails	104	2	48	✓	2819.93		1300	
5	Youtube Extractor: JUnit4	testEncipheredVideo	4	2	24	✓	112.136	✓	9	✓
6		testUnembeddable	7	3	22		151.783		103	
7		testAgeRestrictVideo	5	3	12		103.136		78	
8		testUsualVideo	5	3	14		107.094		77	
9	MyExpenses: Espresso	testScenarioForBug5b..	861	2	100	✓	583.922	✓	704	
10		editCommandKeeps..	101	2	54	✓	929.601		857	
11		cloneCommandIncreases..	–	–	–	–	949.274		879	
12		changeOfFractionDigits..	991	2	170	✓	672.984		693	
13		changeOfFractionDigitsWith..	991	2	174	✓	589.684		637	
14	Firefox Lite: Espresso	saveImageThenDelete..	627	5	352	✓	1954.07	✓	1088	✓
15		dismissMenu	165	2	76	✓	1381	✓	742	
16		turnOnTurboMode..	89	2	40	✓	926.728	✓	5	✓
17		changeDisplayLang	189	3	72	✓	3056.51	✓	680	✓
18	AntennaPod: Robotium	testGoToPreferences	107	3	56	✓	418.265		4	✓
19		testClickNavDrawer	132	2	60	✓	2279.01	✓	5	✓
20		PlaybackSonicTest#..On..	177	10	691	✓	3243.652	✓	1128	
21		PlaybackSonicTest#..Off..	176	6	495	✓	1233.66		1231	
22		PlaybackTest#..Off..Episodes	175	2	142	✓	1251.95		1237	
23		PlaybackTest#..On..Episodes	162	2	140	✓	1071.3		1128	
24	FlexBox: Espresso	testScrollToPosition..row	219	6	306		416.673		837	
25		testAddViewHolders..	46	2	17	✓	142.923		196	
26		testChangeAttributes..	23	2	20	✓	102.37	✓	3	✓
27		testMinHeight..minHeight	58	7	112	✓	105.134		176	
28		testJustifyContent..views	58	2	40	✓	110.923		168	
29		testJustifyContent_center	58	9	158	✓	211.181		173	
30		testFlexWrap..column	55	2	32	✓	107.355		180	
31		testFirstViewGone..column	58	2	32	✓	106.491		168	
32		testChangeOrder..Params	26	4	36	✓	102.21	✓	178	
33		testAlignItems..column	58	2	32	✓	107.798		180	
34		testAlignContent..column	58	2	32	✓	107.021		189	
35		testAlignContent..column	58	2	32	✓	105.902		198	
36		testAlignContent..Padding	56	2	42	✓	107.256		193	
37		testFlexLines..row	74	3	32	✓	246.254		344	
38		testFlexLines..column	69	2	36	✓	221.768		366	
39	Firefox Lite(★): Espresso	browsingWebsite..	642	2	94	✓	2192.65	✓	159	✓
40		saveImageThenDelete..	661	3	96	✓	1415.6		1117	
41	BackPack: JUnit4	test_with_description	41	2	11	✓	146.354		289	
42		test_with_title	40	2	11	✓	150.643		303	
43		test_bottom_sheet_style	32	4	30		149.486		207	
44		test_alert_style	32	4	31		150.811		199	
45		screenshotTestDialog..	86	2	23	✓	285.117		267	
46		screenshotTestDialog	86	2	23	✓	290.37		263	
47		test_with_buttons	42	3	11	✓	174.781		348	
48	Barista: JUnit4	overflowMenuClick_byTitle	130	3	82	✓	240.39	✓	309	
49		openOverflowMenu_..Option	58	2	21	✓	310.67	✓	323	
50		overflowMenuClick_byId	127	4	82	✓	280.342	✓	350	
51	Kaspresso: JUnit4	CommonFlakyTest#test	93	2	26	✓	2232.77		1896	
52		UiCommonFlakyTest#test	94	5	295	✓	2048.74		1941	
Avg/SUM			169	3	101	45	861	15	497	8

test 41, 42, and 47 from app BackPack (11 seconds). *FlakeScanner* detected a flaky test within 3 test runs on average. The maximum number of test runs is for test 20 from AntennaPod (10 runs). To understand why *FlakeScanner* successfully detected failures within a few test runs, we inspected source code of projects. The major reason is that tests heavily use synchronization operations. For instance, test 2 from Surveyor contains 21 test statements, 15 of which use the synchronization operation provided by Espresso [2] `onView()`. When `onView()` is called in the test statement, the testing thread waits until background threads complete tasks. As designed, *FlakeScanner* does not perform event scheduling for those cases since async events are bounded in the execution of a test statement by the synchronization operation. Then *FlakeScanner* can focus on scheduling events for statements that lack synchronization operations. Moreover, *FlakeScanner* prioritizes exploring positions that are closer to the upper bound event, which likely triggers flaky test failures (Section 4.3). Thus, *FlakeScanner* could detect failures within a few test runs. The results also show *FlakeScanner* is a practical tool. It successfully detected flaky test failures for test 12 and 13 from MyExpenses, for which 991 events were generated in the execution. Meanwhile, *FlakeScanner* worked on Android projects that adopt different testing frameworks such as Espresso [2] and Robotium [3].

FlakeScanner successfully detected 45 out of 52 known flaky tests in 10 Android projects. On average, it detected a flaky test within 3 test runs.

7.5 RQ2: Comparison with Existing Techniques

As shown in Table 4, *FlakeScanner* outperforms *Shaker* and *100RUN* in terms of both the number of detected flaky tests and the average execution time. Out of the 52 known flaky tests, *FlakeScanner* detected the most flaky tests (45) and is followed by *Shaker* (15) and *100RUN* (8). For execution time, *FlakeScanner* detected a flaky test in 101 seconds on average, which is less than 861 seconds of *Shaker* and 497 seconds of *100RUN*. For most of tests, *100RUN* could not detect a flaky test failure in the first few runs and kept executing them until reaching 100 times, which took a longer time.

Regarding the overlap between flaky tests detected by each tool, *FlakeScanner* detected all the flaky tests detected by *Shaker* and *100RUN*. But *Shaker* and *100RUN* failed to detect the other flaky tests that were detected by *FlakeScanner*. The better results from *FlakeScanner* can be explained as follows: *FlakeScanner* can identify synchronization operations in the test execution and focus on scheduling events for statements that lack synchronization operations. Unexpected event execution orders that cause flaky test failures are more likely to explored by *FlakeScanner*.

We also evaluated *EventRacer* since it uses dynamic analysis to infer event race for Android apps. *EventRacer* reported many possible races for each test run. On average, it reported 1237 races for a test. This is because *EventRacer* focuses on detecting races in Android apps and does not analyze races in which testing frameworks are involved. Furthermore, *EventRacer* infers races by analyzing recorded traces and cannot validate whether the reported races can cause flaky test failures.

FlakeScanner outperforms *Shaker* and *100RUN* in terms of both the number of detected flaky tests and average execution time.

7.6 RQ3: Real-world Flaky Test Detection

We ran 1444 passing tests from the 33 Android projects in *FlakyAppRepo* which are not annotated as flaky tests (these tests may or may not be flaky). Out of these 33 projects, *FlakeScanner* detected at least one flaky test for 19 projects, and reported 245 flaky tests in total. To validate previously unknown flaky tests that *FlakeScanner* detected, we randomly selected 20 out of the detected 245 flaky tests and reported them to developers. For each selected test, we manually reproduced the failure that *FlakeScanner* witnessed during detection and generated a detailed root-cause-analysis report, and submitted the report on the Github. At the time of writing the paper, we got responses on 15 test cases. Out of the 15 tests, 13 were confirmed as flaky tests and addressed by developers. For the other two tests, developers replied that the reported failures were not encountered yet or not reproduced at their end, without giving us further explanation. Our experience with flaky test reporting shows so far that developers are more interested in identifying which tests are flaky. Once a flaky test is detected, they appear to be more prone to removing them, rather than investigating why it is flaky.

FlakeScanner detected 245 previously unknown flaky tests in 19 widely-used Android projects. Out of the reported 20 unknown flaky tests, 13 were confirmed and addressed by developers.

7.7 Threats to Validity

External Validity: Threats to external validity relate to the generalizability of the experimental results. *FlakeScanner* is evaluated so far on 33 Android projects. Our results may not generalise beyond the 33 Android projects to which we have applied *FlakeScanner*. To mitigate this threat, we not only choose Android projects that are popular and well maintained but also include less popular Android projects (i.e., less stars on the Github) which were searched on the Github via keywords.

Internal Validity: Threats to internal validity concern factors in our experimental methodology that may affect our results. In Study 1, we note that 52 concurrency or synchronization related known flaky tests are chosen by manually analyzing their related descriptions and commit messages, which might result in selection bias. Similarly, we manually analyze failures detected by each tool under evaluation and validate the results, which might introduce bias as well. To mitigate these risks, two authors of this paper independently performed the manual tasks, and cross-checked each other's results.

7.8 Data Availability

To facilitate future research on flaky tests, we make our prototype *FlakeScanner* and subject-suite *FlakyAppRepo* available at link https://github.com/AndroidFlakyTest

8 RELATED WORK

Flaky Test Detection. Bell et al. [6] use code coverage based differential analysis to identify flaky tests. A test is deemed flaky if it fails during regression testing and its execution does not reach any code that was recently changed by developers. Lam et al. propose iDFlakies [27], a framework for detecting order dependent flaky tests, tests that fail when run in different orders. Dutta et al. [13] develop an approach to detect random number related flaky tests, tests that fail due to difference in the sequence of random numbers generated in different runs. Shi et al. [39] propose an approach to fix order-dependent flaky tests by leveraging passing tests. Shi et al. [37] propose to rerun a test multiple times on each mutant and obtain reliable coverage results such that the effects of flaky tests on mutation testing can be mitigated. In contrast, *FlakeScanner* detects concurrency or synchronization related flaky tests in Android projects by exploring feasible event execution orders.

Event Race Detection. Another branch of works that are close to ours is event race detection [8, 18–21, 31–33, 35, 36, 45]. Instead of detecting flaky tests, these works leverage dynamic and static analysis to detect event races. For instance, ER Catcher [35], DROIDRACER [32], *EventRacer* [8], CAFA [19], and nAdroid [18] capture *happens-before-relation* among events and infer possible event races. In addition, Ozkan et al. [24] propose to detect asynchronous bugs by exploring different execution orders of event handlers in Android apps. SARD [46] leverages happens-before analysis to detect use-after-free issues in Android apps. These techniques have the potential to be applicable for flaky test detection, but face challenges to capture complete and precise happens-before relations when a test is executed by a testing framework. Many false positives can be reported by event-race detectors due to incomplete happen-before relations being compute. In contrast, *FlakeScanner* performs a system-level dynamic analysis to capture precise event dependencies to avoid such false positives.

Empirical Studies on Flaky Tests. Multiple studies [14, 26, 30, 42] confirm concurrency as the major cause of flaky tests. Luo et al. [30] performed an empirical analysis of flaky tests in 51 open-source projects. They identified *Concurrency* and *Async wait* as the most common cause of flaky tests. They pointed out that the majority of these cases arose because they do not wait for external resources. Finally, they described the common fixing strategies the developers use to fix flaky tests. In a separate study, Eck et al.[14] surveyed 21 professional developers to classify 200 flaky tests they fixed. They identified four unreported causes of flaky tests, which are also considered difficult to fix. Thorve et al. [42] conducted an empirical study of flaky tests in Android apps. They searched 1000 projects for the commits related to flakiness and found only 77 relevant commits from 29 projects. They found 36% of commits occurred due to concurrency related issues. Fan et al. [16] proposed a hybrid approach towards manifesting asynchronous bugs in Android apps with fault patterns.

Concurrency Bug Detection. There have been several testing based approaches [10, 12, 22, 23, 28, 47] to identify concurrency related bugs. Maple [47] proposed a coverage-driven approach to expose untested thread interleavings. Letko [28] proposed a combination of testing and dynamic analysis with *metaheuuristic* techniques. Choudhary et al. [10] presented a coverage-guided approach for generating concurrency tests to detect bugs in thread-safe classes. Multiple related works [5, 7, 9, 15, 29, 43] manipulated event orders to control non-determinism in multi-threaded programs. Liu et al. [29] proposed a deterministic multithreading system that replaces *pthreads* library in C/C++ apps. Emmi et al. [15] proposed a search prioritization strategy to discover concurrency bugs. They add non-determinism to deterministic schedulers by delaying their next-scheduled task. Adamsen et al. [5] presented an automated program repair technique for event race errors in JavaScript. Given a repair policy, they controlled the event handler scheduling in the browser to avoid bad orderings.

9 DISCUSSION

Flaky tests pose a significant problem in validating mobile apps. In this paper, we presented an approach for detecting flaky tests via systematic event order exploration. We introduced *FlakeScanner*, a tool to detect flaky tests for Android apps. *FlakeScanner* explores the space of possible execution environments which may cause relevant threads to interleave differently. Due to the lack of a testing benchmark for flaky tests, we created the first subject-suite *FlakyAppRepo* that is used to study test flakiness. *FlakyAppRepo* contains 33 widely-used Android apps with around 2.5k stars on average in GitHub. We applied *FlakeScanner* to tests from *FlakyAppRepo*. Results show that *FlakeScanner* not only detected known flaky tests but also reported 245 new flaky tests. We believe that our tool and results hold out promise for tackling flaky tests, which is a significant pain point in the practice of testing.

ACKNOWLEDGEMENTS

This work was partially supported by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems).

REFERENCES

[1] 2016. Flaky Tests at Google and How We Mitigate Them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html
[2] 2020. Espresso. https://developer.android.com/training/testing/espresso
[3] 2020. Robotium. https://github.com/RobotiumTech/robotium
[4] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions.
[5] C. Q. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, and K. Sen. 2017. Repairing Event Race Errors by Controlling Nondeterminism. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.
[6] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*.
[7] Tom Bergan, Luis Ceze, and Dan Grossman. 2013. Input-covering schedules for multithreaded programs. *ACM SIGPLAN Notices* (2013), 677–692.
[8] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. *ACM SIGPLAN Notices* (2015).
[9] Ahmed Bouajjani, Michael Emmi, Constantin Enea, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. 2017. *Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency.*
[10] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
[11] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[12] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.fz: Fuzzing the Server-Side Event-Driven Architecture. In *European Conference on Computer Systems (Eurosys)*.

[13] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[14] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[15] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-Bounded Scheduling. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*. 411–422.

[16] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently Manifesting Asynchronous Programming Errors in Android Apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[17] Martin Flower. 2020. Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html

[18] 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 (CGO)*.

[19] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano Pereira, Gilles Pokam, Peter Chen, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. *ACM SIGPLAN Notices* (2014).

[20] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically verifying and reproducing event-based races in Android apps. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[21] Jeff Huang and Arun K. Rajagopalan. 2016. Precise and Maximal Race Detection from Incomplete Traces. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[22] Casey Klein, Matthew Flatt, and Robert Findler. 2010. Random Testing for Higher-Order, Stateful Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. 555–566.

[23] Bohuslav Krena, Zdenek Letko, Tomas Vojnar, and Shmuel Ur. 2010. A platform for search-based testing of concurrent software. In *PADTAD 2010 - International Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*.

[24] Burcu Kulahcioglu Ozkan, Michael Emmi, and Serdar Tasiran. 2015. Systematic Asynchrony Bug Exploration for Android Apps. In *International Conference on Computer Aided Verification (CAV)*.

[25] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[26] Wing Lam, Kivanc Muslu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *42nd International Conference on Software Engineering*.

[27] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *12th IEEE Conference on Software Testing, Validation and Verification*.

[28] Zdeněk Letko. 2013. Analysis and Testing of Concurrent Programs. *Information Sciences and Technologies Bulletin of the ACM Slovakia* (2013).

[29] Tongping Liu, Charlie Curtsinger, and Emery Berger. 2011. Dthreads: Efficient deterministic multithreading. In *SOSP'11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 327–336.

[30] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering (FSE)*.

[31] Pallavi Maiya and Aditya Kanade. 2017. Efficient Computation of Happens-before Relation for Event-Driven Programs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[32] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. [n.d.]. Race Detection for Android Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[33] Arun K. Rajagopalan and Jeff Huang. 2015. RDIT: Race Detection from Incomplete Traces. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*.

[34] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An Empirical Analysis of UI-based Flaky Tests. In *IEEE/ACM International Conference on Software Engineering*.

[35] Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. ER Catcher: A Static Analysis Framework for Accurate and Scalable Event-Race Detection in Android. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*.

[36] Anirudh Santhiar, Shalini Kaleeswaran, and Aditya Kanade. 2016. Efficient Race Detection in the Presence of Programmatic Event Loops. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

[37] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[38] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[39] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: a framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC-FSE)*.

[40] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *IEEE International Conference on Software Maintenance and Evolution*.

[41] Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. 2020. A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*.

[42] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *International Conference on Software Maintenance and Evolution (ICSME)*. 534–538.

[43] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Verifying Concurrent Programs by Memory Unwinding. In *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[44] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*.

[45] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue. 2019. Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*.

[46] Diyu Wu, Jie Liu, Yulei Sui, Shiping Chen, and Jingling Xue. 2019. Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*.

[47] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*.

[48] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*.