Reproducing Timing-Dependent GUI Flaky Tests in Android Apps via a Single Event Delay

Xiaobao Cai
caixb22@m.fudan.edu.cn
Fudan University
Shanghai, China

Zhen Dong*
zhendong@fudan.edu.cn
Fudan University
Shanghai, China

Yongjiang Wang
yjwang23@m.fudan.edu.cn
Fudan University
Shanghai, China

Abhishek Tiwari
abhishek.tiwari@uni-passau.de
University of Passau
Passau, Germany

Xin Peng
pengxin@fudan.edu.cn
Fudan University
Shanghai, China

Abstract

Flaky tests hinder the development process by exhibiting uncertain behavior in regression testing. A flaky test may pass in some runs and fail in others while running on the same code version. The non-deterministic outcome frequently misleads the developers into debugging non-existent faults in the code. To effectively debug the flaky tests, developers need to reproduce them. The industry de facto to reproduce flaky tests is to rerun them multiple times. However, rerunning a flaky test numerous times is time and resource-consuming.

This work presents a technique for rapidly and reliably reproducing timing-dependent GUI flaky tests, acknowledged as the most common type of flaky tests in Android apps. Our insight is that flakiness in such tests often stems from event racing on GUI data. Given stack traces of a failure, our technique employs dynamic analysis to infer event races likely leading to the failure and reproduces it by selectively delaying only *relevant* events involved in these races. Thus, our technique can efficiently reproduce a failure within minimal test runs. The experiments conducted on 80 timing-dependent flaky tests collected from 22 widely-used Android apps show our technique is efficient in flaky test failure reproduction. Out of the 80 flaky tests, our technique could successfully reproduce 73 within 1.71 test runs on average. Notably, it exhibited extremely high reliability by consistently reproducing the failure for 20 runs.

CCS Concepts

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

Keywords

Regression Testing, Event Racing, Dynamic Analysis, Failure Reproduction

ACM Reference Format:
Xiaobao Cai, Zhen Dong, Yongjiang Wang, Abhishek Tiwari, and Xin Peng. 2024. Reproducing Timing-Dependent GUI Flaky Tests in Android Apps via a Single Event Delay. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680377

1 Introduction

Testing is an indispensable part of software development; tests are run to find faults in the code being changed or developed. Unfortunately, software development suffers from *flaky tests*. A flaky test impedes the software development process by passing in some runs and failing in others while running on the same codebase version [19]. In other words, a test failure may not indicate a fault in the code. A test failing due to flakiness can mislead developers to debug non-existent bugs in the code, slowing down the development cycle. Besides, flaky tests are often hard to reproduce [5, 24]. This problem has been identified as a significant obstacle in software development at large companies such as Facebook [11], Google [22, 32], and Microsoft [14, 15].

Reproducing flaky test failures is often challenging yet necessary. The error messages and stack traces of a flaky test failure indicate only the symptom, such as a timeout error or assertion failure, without obvious hints pointing to the underlying root cause. To debug such flakiness, a developer needs to repeatedly reproduce a failing execution so that she/he can enable extensive logging and observe runtime states accordingly. In fact, recent work [10] shows that 77% of developers often run flaky tests multiple times when debugging a flaky-test failure to log different parts of code and analyze its runtime behavior. Unfortunately, such reproduction is non-trivial since most flaky tests fail rarely or only in a specific execution environment, particularly for concurrency-related flaky tests. Addressing this challenge, Leesatapornwongsa et al., in Microsoft [18], develop an approach to reproduce concurrency-related flaky test failures in .NET projects automatically. Lam et al. [17] explore the effectiveness of rerun in reproducing flaky test failures in Java projects.

*Corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680377

While some approaches [8, 28] detect flaky tests in Android, unfortunately, none guide reproducing them. Reproducing flaky deterministically helps developers understand and debug them better. This paper introduces an approach to automatically reproducing flaky test failures in Android apps. Unlike traditional programs, Android adopts the event-driven model in which the execution flow is determined by events such as user interactions (e.g., button clicking) and sensor signals. On the Android platform, tests are executed to simulate events to exercise the app under test. However, they are often flaky due to timing-related issues, i.e., an event is executed at an incorrect timing due to asynchronous waits that might take too long or an event occurring earlier than expected. This type of flaky tests are called *timing-dependent flaky tests* and have been identified as the most prominent flaky tests in Android apps according to recent studies [25, 30]. Therefore, we focus on timing-dependent flaky tests failure reproduction. Given such a flaky test and its failure symptom (i.e., error messages and crash stack traces), our approach *deterministically* reproduces a failed execution that generates the same symptom.

We aim to develop a practical approach to reproduce flaky test failures efficiently. Our approach must explore event execution orders intelligently; the number of possible event execution orders grows exponentially with the number of events generated in a test run. Existing event race detection tools, such as CAFA [12], DroidRacer [21], and ERVA [13], narrow down the exploration space by tracking fine-grain *happens-before relations*, which require instrumenting all the read and write operations. Unfortunately, it is hard to scale them to large apps. Recent flaky test detection tools such as Shaker [28] explore possible event orders by adding noise in the execution environment. Unfortunately, these approaches tend to explore a large number of event execution orders per test. To ensure efficiency, we need to find a solution that can manifest failures by exploring minimal event orders.

Our insight is that flakiness in GUI flaky tests often stems from event racing on GUI data. We can infer event races likely leading to the given failure during a test run and reproduces it by selectively delaying only *relevant* events involved in these races. Therefore, we can efficiently reproduce a failure within minimal test runs. To achieve this, we employ dynamic analysis to identify event racing on GUI data and deem event races that occur prior to the given failure as "root cause" event races. Then, we can reproduce the failure by delaying only events involved in these event races.

To identify event races prior to the given failure, we generate a flame chart [2] that records call stacks throughout a test run, enabling it to identify a potential point where the given failure may occur based on its crash stack traces and the corresponding events leading up to that point. To identify event racing on GUI data, we extract a list of methods accessing GUI widgets such as `TextView` from the Android documentation and monitor these methods at runtime. To delay an event in a test run, we leverage a key observation related to the Android event-driven concurrency architecture. In this architectural model, threads in the Android framework communicate with each other by posting events through a designated method called `enqueueMessage()`. By dynamically hooking this method at runtime, we gain the ability to record events generated during a test run and selectively delay the execution of specific events. This allows us to dynamically explore and manipulate event orders without requiring any code instrumentation or modifications in the app under test and the Android framework.

We implemented our approach into a fully automated tool called FlakeEcho and conducted evaluations on 80 timing-dependent flaky tests in 22 widely used and large open-source apps from GitHub. The experiment results show the effectiveness of FlakeEcho in reproducing timing-dependent flaky tests in Android apps, successfully reproducing 73 out of the 80 flaky tests. Notably, all the 73 flaky tests can be successfully reproduced by delaying a single event. On average, it takes 1.71 test runs to reproduce a failure. Additionally, for each failure, FlakeEcho generates a delay configuration that allows the failure to be reproduced deterministically. Under this delay configuration, the failure manifests in 16.25 out of 20 runs. In summary, our work makes the following contributions:

- **Practical Technique.** We present an event delay-based technique for reproducing timing-dependent flaky test failures in Android apps. Our approach can efficiently reproduce flaky tests within a few test runs. The key novelty of our tool, FlakeEcho, lies in inferring the "root cause" event race of a given failure and reproducing it by selectively delaying only relevant events, consequently avoiding exhaustedly exploring all possible event execution orders. More importantly, for each failure, our approach generates a delay configuration that allows the failure to be reproduced reliably.
- **Extensive Evaluation.** We evaluated FlakeEcho on 80 timing-dependent flaky tests from 22 popular open-source Android apps. We measured the efficiency of the delay configurations generated by FlakeEcho in reliably reproducing the given failures over 20 test runs. To support further research on flaky test reproduction, we have made FlakeEcho and the dataset publicly available at: https://github.com/FlakeEcho/FlakeEcho.

2 Background and Scope

This section provides a brief overview of the event-driven programming model in Android. Next, we explain the timing-dependent flaky tests that are the main focus of our work and present an example case.

Event-driven Concurrency Model. In Android's concurrency model, every app process has a main thread (called *UIthread*) and several *background* threads. To achieve rapid UI responsiveness, only the UI thread can access GUI objects. To update the GUI, background threads send events to the UI thread, and the UI thread dispatches these events to the appropriate UI widgets. Long-running tasks, such as network access, usually run in background threads. When these tasks are completed, the background threads post events together with the data to the UI thread. This concurrency model involves three main components:

- **Event.** Events can be generated by external entities or internally by threads or events executed within the application. They originate from various sources, including input devices like screen and GPS sensors, the Android framework (e.g., low battery notifications), third-party libraries (e.g., Google services), and app components (e.g., threads exchanging data).

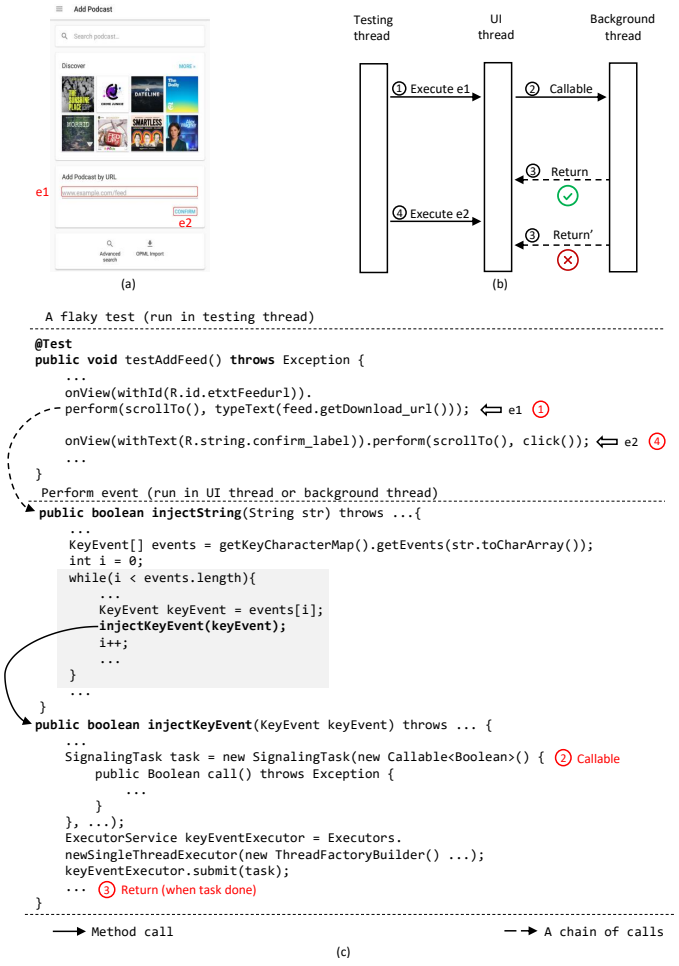


Figure 1: A timing-dependent flaky test in Android app Antennapod.

- **Event Queue.** Once an event is generated, it is placed in an event queue. Events in the queue are processed in the order they were queued, ensuring sequential handling.
- **Looper Thread.** A looper thread is associated with each event queue. Its role is to continuously check its corresponding event queue and process events one by one. This ensures that all events executed in a looper thread are atomic with respect to each other. In Android, the UI thread functions as the looper thread.

Timing-dependent Flaky Tests. This paper refers to tests that yield varying outcomes based on specific execution orders as timing-dependent flaky tests. These tests pass or fail depending on the timing or sequencing of events during execution. Two common patterns in timing-dependent flaky tests are (1) *async waits*. In this scenario, a test starts an asynchronous operation and instead of using proper synchronization to ensure its completion, it waits for a specific period and assumes that the asynchronous operation has been completed within that period. The test fails when the operation does not complete within the fixed wait period.; (2) *data race*. Another common pattern is data race caused by multiple threads accessing the same object instance without proper synchronization. A data race can non-deterministically lead to test failure. They have been identified as the most prominent flaky tests in Android apps according to recent studies [25, 30].

Example. Figure 1 shows a timing-dependent flaky test in Android app *Antennapod*, a popular open-source podcast player (5.4k stars on Github). The test is shown in Figure 1 (c) and designed to validate the feed-adding functionality on the screen in Figure 1 (a). It first localizes the textbox widget on the screen and inputs a predefined URL in the textbox, and then clicks the "CONFIRM" button to add the feed to the app. Afterward, it checks if the feed is successfully subscribed.

This test is processed as follows. As shown in Figure 1 (b), it first is loaded into a dedicated testing thread that interacts with the app under test through events. When the first statement is executed, it generates an event ①. Receiving event ①, the UI thread processes the event and starts to type the given URL into the textbox. Considering typing URL involves operating the keyboard, the UI thread offloads the task to a background thread (②). As shown in the method `injectKeyEvent()`, each key operation is performed by the background thread. Once the key operation is completed, the background thread sends an event (③) to the UI thread for updating the corresponding letter into the textbox. This process is repeated until all the letters in the URL are typed into the textbox. Once the URL typing is done, the testing thread executes the second statement, sending event ④ to the UI thread. Then the UI thread adds the URL to the specified location.

The outcome of the test depends on the execution timing of event ④, i.e., "CONFIRM" button clicking. It passes if event ④ is executed after the URL being completely filled into the textbox. Otherwise, it fails. Let us assume event ③ is the last event that is generated by the background thread in the URL typing task. As shown in Figure 1 (b), if the event ④ occurs after ③, the test passes. Otherwise, the test fails. In fact, this order cannot be guaranteed due to the lack of a reliable synchronization mechanism between the testing thread and the background thread. Thus, the test manifests flaky behavior, passing for some runs and failing for others.

3 Overview of Our Approach

This work proposes an approach to reproduce timing-dependent flaky test failures within minimal runs. The key observation is that such failures often arise from event races on GUI data. The test passes for certain event execution orders but fails for specific ones. To manifest such failures effectively, we apply the following two heuristics in our approach:

- Event races are more likely to lead to a given failure if they race on GUI data [25].
- Event races that occur before the failure point are more likely to lead to a given failure.

Based on these two heuristics, our approach identifies GUI data-related event races that occur in the test run and infers event races that are more likely to trigger the failure. It then prioritizes the exploration of event orders caused by these races.

Problem Formulation. To formulate our problem, we leverage the event-driven concurrency model established by Hu et al. [13]

Thread	t	::=	$UIThread$
		|	$BackgroundThread$
		|	$TestingThread$
Thread operation	γ	::=	$fork(t_1, t_2)$
		|	$join(t_1, t_2)$
Memory location	ρ	$\in$	$Pointers$
Memory access	α	::=	$\alpha_\tau(\rho)$
Access type	τ	::=	$read$
		|	$write$
Event	e	::=	$begin; op_1; ...op_n; end$
Event type	v	::=	$ExternalEvent$
		|	$RunnableObject$
		|	$Message$
Event posting	β	::=	$post(e, t_1, t_2, \Delta)$
Operation	op	::=	α
		|	β
		|	γ
Trace	π	::=	$op_1; ...; op_n$

Figure 2: Event-Driven Concurrency Model

(Figure 2). In this model, threads t can generally be categorized as either the UI thread, a testing thread, or a background thread. The *Thread operation* γ can either fork a new thread from an existing one or join a thread into another. Event e can be an external event that may originate from user interactions (input event) or the Android system (system event). Besides, it can also be a *message* or *runnable* object that is usually sent from a background or testing thread to the UI thread for communication. For example, when a task is done, the background thread sends a *runnable object* to return the results to the UI thread. Memory access $\alpha_\tau(\rho)$ can read or write memory location ρ. The event posting operation generates and dispatches an event with a $Delta$-second latency from one thread to another. Generally, a background or testing thread posts events to the UI thread to update results, and Δ is set to 0 by default. Trace π contains a sequence of operations above.

Happens-before Relation. We define a happens-before relation, similar to the traditional concurrency model, as the minimum partial order (i.e., reflexive, anti-symmetric, transitive relation) over events of a trace such that $e_1 \prec e_2$ if :

- *Program Order Rule:* $end(e_1) \prec begin(e_2) \wedge e_1 \in post(t) \wedge e_2 \in post(t)$
- *Thread Rule:* $end(e_1) \prec begin(e_2) \wedge e_1 \in post(t_1) \wedge e_2 \in post(t_2) \wedge fork(t_1, t_2)$

Two unordered events e_i and e_j are denoted as $e_i \parallel e_j$, if they are not related by the happens-before relation.

Definition 3.1 (Event Races). An event e_i races with another event e_j if there exists a shared variable ρ such that $\alpha_i(\rho) \in e_i, \alpha_j(\rho) \in e_j$ and $e_i \parallel e_j$, and at least one of $\alpha_i(\rho)$ and $\alpha_j(\rho)$ is a write.

Timing Flaky Test Failure Reproduction. Timing-dependent flaky test reproduction can be addressed as an event order exploration problem. Given a test T and its failure F, the objective is to find an event sequence $\overrightarrow{E} \in \Omega(Z, \prec)$, where $\overrightarrow{E}$ is the event execution order when the failure F occurs; Z indicates the set of events generated during the execution of test T. The space of possible event execution orders $\Omega(Z, \prec)$ becomes extremely huge as the number of generated events increases. Blindly exploring such a space is challenging and impractical.

Our Idea. We tackle the challenge by leveraging the key insight that timing-dependent flaky tests often result from event racing on GUI data during test execution. Our approach involves analyzing GUI-related event races in the test run and inferring event races that may occur before the given failure. We only explore event orders caused by these races, enabling a more efficient manifestation of timing-dependent flaky test failures.

4 Detailed Approach

The workflow of FlakeEcho is depicted in Figure 3. FlakeEcho takes the app under test and the test case as input to perform a dynamic analysis. During the analysis, FlakeEcho monitors and records events during execution and GUI access operations. Besides, it captures the execution traces of a test run. Afterward, the collected runtime data and crash stacks are analyzed to identify event races that could have caused the failure. Subsequently, FlakeEcho runs the test by enforcing an order among the events involved in the race to manifest the failure. To establish an order between two events, FlakeEcho delays one event to allow the other to execute first. If the test run successfully reproduces the observed flaky behavior, FlakeEcho provides the delay configuration as output.

4.1 Event Tracing and GUI Access Analysis

Event Identification. To analyze the event races during a test execution, FlakeEcho first records the generated events. As explained in the background, Android follows an event-driven concurrency model, where the events that occur during execution are added to the event queue and processed in sequential order. Consequently, we record events by hooking the API that places the events into the queue. Specifically, the event queue object provides a method `enqueueMessage()` for posting an event. When an event occurs, the method will be invoked to place it into the queue. To prevent any modifications to the Android system and the app under test, FlakeEcho dynamically hooks this method to record the events.

Unfortunately, the events captured at runtime do not have unique identifiers. In fact, the captured events are objects that contain dynamically generated data. These objects serve as containers for information that needs to be passed to the event handlers, and they are often recycled in the execution process. After an event is processed, it is returned to a pool for reuse later, avoiding unnecessary object creation [1]. Identifying an event with the content in the event object is challenging. On the other hand, FlakeEcho requires identifying events generated in the test execution, analyzing races among them, and delaying the execution of an event in the test run. Thus, we need to develop an approach to generate an event identifier that can be used across test runs.

To tackle this challenge, we leverage the method employed by FlakeRepro [18] to compute a unique ID for each event (event ID) that can be consistently used during test runs. The event ID is derived by combining the consistent ID of the thread that posts the event with the *execution context* within that thread. Specifically,

[1] https://developer.android.com/reference/android/os/Message

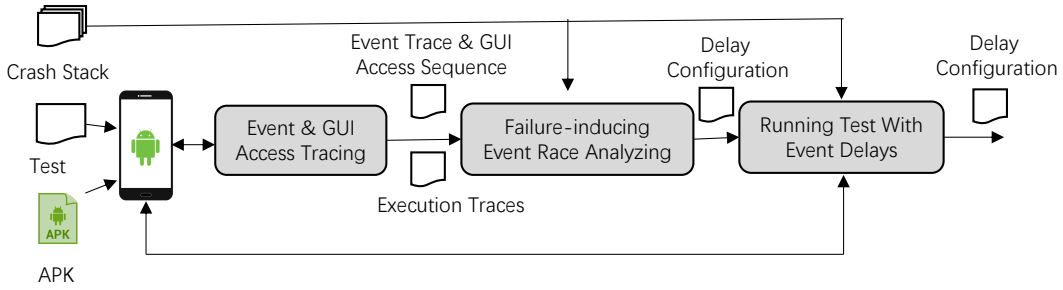


Figure 3: The workflow of our approach FlakeEcho

```
android.preference.EditTextPreference: void setText()
android.preference.EditTextPreference: java.lang.String getText()
android.app.ProgressDialog: void setMax()
android.app.ProgressDialog: int getMax()
android.preference.SwitchPreference: void setSwitchTextOff()
android.preference.SwitchPreference: java.lang.CharSequence getSwitchTextOff()
...
```

Figure 4: A list of methods that access GUI data in Android

the event ID comprises (1) the consistent ID of the thread posting the event, (2) the call stack of the thread at the time of posting the event, and (3) the number of times the thread posts an event with the same call stack. The consistent thread ID needs to remain the same across runs. However, the thread ID assigned by the underlying runtime may vary, as the same logical thread may have different IDs across different runs. To overcome this issue, we leverage the observation that threads created in the Android system are often given a thread name that remains consistent across runs. Therefore, FlakeEcho utilizes the thread name as the consistent thread ID. By obtaining this data at runtime, FlakeEcho can compute the event ID and record events that occur during the test run.

GUI Access Analysis. To effectively analyze event races, it is essential to track both writing and reading operations during event processing. Ideally, all data access operations should be recorded, as event races can arise when two distinct events concurrently read or write to shared variables in memory. However, in the scope of this work, we do not need to record all data access operations: Our main focus is to identify event races that have the potential to cause timing-dependent flaky test failures, instead of detecting general concurrency bugs in the app. With GUI testing as the main focus, it is common for test failures to happen because of differences between expected results and the data retrieved from the GUI. Therefore, we consider only event races on GUI data in this work.

To effectively identify GUI data access operations in Android apps, we begin with a review of the Android documentation. Since user interfaces are constructed using `View` and `ViewGroup` objects in Android, we focus on examining methods provided by these classes and their subclasses, such as `TextView` and `LinearLayout`. Figure 4 shows a list of methods that access GUI data in Android. This analysis allows us to compile a comprehensive list of GUI reading and writing operations. To record GUI accesses triggered by an event, we leverage the event dispatching mechanism in Android that events are dispatched via method `dispatchMessage()`. The task can be completed in two steps:

- Given an event e, FlakeEcho identifies when the event e starts to be processed and when the processing is completed by hooking method `dispatchMessage()` at runtime;
- During processing event e, FlakeEcho tracks GUI operations by hooking the compiled GUI access methods, recording which property values of widgets on the screen are read or written.

Specifically, for event e, FlakeEcho records its ID and a set of variables in the GUI widgets that are read or modified, which are denoted by $\langle ID, \{\alpha_\tau(\rho_1), ..., \alpha_\tau(\rho_n)\}\rangle$, where τ indicates the access type (i.e., read or write) and ρ indicates accessed variables.

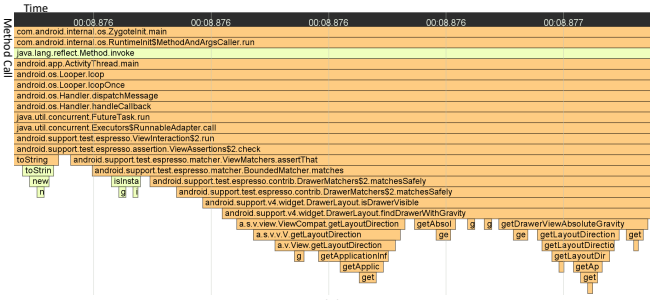
Profiling. To generate a flame chart for a test run, FlakeEcho leverages an open-source tool Android Method Profiler that can record call stacks over a test run and analyze traces. It is similar to the official Android Profiler but operates quickly.

4.2 Failure-inducing Event Race Inference

Given the execution trace and events collected in the previous step, along with the information about the given failure, we can infer event races that may lead to the failure. The key idea is to identify the point in the execution trace at which the given failure may occur and then analyze the races among events that occur before the point. These event races are considered failure-inducing races. Before diving into details, we first introduce two crucial data structures used in the approach:

- Flame Chart. A flame chart is a visualization that showcases stack samples chronologically during an app's runtime. Each rectangle in the chart corresponds to a stack frame, representing a specific method or function executed during the profiling. Figure 5 (a) shows an example of flame chart.
- Crash Stack. The call stack at the time of a failure is referred as the crash stack shown in Figure 5 (b). It provides a snapshot of the functions or methods that were being executed in the app when the failure occurred.

Algorithm 1 outlines the process of inferring failure-inducing event races. It takes as input (1) the event sequence and execution



(a)

```
junit.framework.AssertionFailedError
    at android.support.test.espresso.matcher.ViewMatchers.assertThat(ViewMatchers.java:1053)
    at android.support.test.espresso.assertion.ViewAssertions$2.check(ViewAssertions.java:89)
    at android.support.test.espresso.ViewInteraction$2.run(ViewInteraction.java:170)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:462)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    ...                                                    Crashing Method
```

(b)

Figure 5: An example of flame chart and crashing stack trace

Algorithm 1: Failure-inducing Event Race Inferring

Input: $\overrightarrow{E}$:Event sequence
Input: π: Execution Trace
Input: s: Crash stack of the given failure
Input: k: Number of suspicious event races

1 **Procedure** `Infer`($\overrightarrow{E}, \pi, s, k$)
2 $\quad\overrightarrow{S} \leftarrow$ `generateFlameChart`(π);
3 $\quad D \leftarrow$ `extractThreadDependence`($\overrightarrow{S}$);
4 $\quad i \leftarrow$ `identifyFailurePoint`($\overrightarrow{S}, s$);
5 $\quad\overrightarrow{E_b} \leftarrow$ `getEventSequenceBeforeFailurePoint`($\overrightarrow{S}, \overrightarrow{E}, i$);
6 $\quad\overrightarrow{R} \leftarrow$ `computeEventRaces`($\overrightarrow{E_b}, D$);
7 $\quad\overrightarrow{R_k} \leftarrow$ `fetchKNearestRaces`($\overrightarrow{R}, k$);
8 $\quad$return $\overrightarrow{R_k}$

trace collected from the dynamic analysis; (2) the crash stack of the given failure; (3) the parameter k specifying the number of suspicious event races that need to be considered in the event order exploration; it can be configured according to scenarios or requirements. As shown in Line 2, FlakeEcho first takes execution trace π to generate a flame chart by invoking the tool Android Method Profiler. The flame chart is a sequence of call stacks over time, which is denoted with $\overrightarrow{S}$. At Line 3, it analyzes the flame chart $\overrightarrow{S}$ to extract the threads involved in the test and compute relations among them, i.e., *fork* and *join* relations (denoted by D).

Possible Failure Point Identification. Line 4 analyzes a point in the flame chart $\overrightarrow{S}$ at which the given failure likely occurs. The study [23] shows stack-matching effectively identifies a re-occurring failure, and the *prefix-matching* algorithm is more precise. Moreover, the stack traces of different test runs may exhibit slight variations. Thus, we use the prefix matching algorithm to compute the similarity between the crash stack s of the failure and each call stack in $\overrightarrow{S}$. The point with the highest similarity is deemed as the possible failure point. Specifically, the similarity between two call stacks s_1 and s_2 is computed with the following formula:

$$argmax_i\{s_1^i : s_1^i = s_2^i\}/max(|s_1|, |s_2|)$$

That is, the similarity score of two call stacks is determined by the size of the longest common prefix normalized to the length of the longer of the two call stacks.

Suspicious Event Races Computation. FlakeEcho identifies event races as suspicious if they involve racing on GUI data and occur prior to the Failure Point. Line 5 to Line 7 of Algorithm 1 explains this workflow. Line 5 performs an analysis to extract the sub-event sequence $\overrightarrow{E_b}$ that occurs prior to the possible failure point identified in the previous step. To achieve this, the algorithm first maps the events in $\overrightarrow{E}$ onto the flame chart $\overrightarrow{S}$. The mapping process is as follows: for each event e_i in $\overrightarrow{E}$, the algorithm compares its call stack with each call stack in $\overrightarrow{S}$. If exactly matched, e_i is located at the corresponding point on the flame chart $\overrightarrow{S}$. Note that there is always an exact match between each event's call stack and the flame chart because the recording of each event's call stack and the flame chart creation happens within the same test execution. After all the events are located on $\overrightarrow{S}$, the algorithm identifies the failure point i on $\overrightarrow{S}$. As recall, a flame chart is a visual representation of call stacks arranged chronologically, therefore, we can identify events that occur prior to the failure point on $\overrightarrow{S}$. Specifically, the algorithm takes the events preceding i on $\overrightarrow{S}$ as the sub sequence of events $\overrightarrow{E_b}$ that occur prior to the failure point.

Line 6 performs a computation to identify GUI-related event races in $\overrightarrow{E_b}$ using the definition of event races described in Section 3. For e_i and e_j in $\overrightarrow{E_b}$, there exists a GUI-related event race between them if e_i and e_j access the same variable value on GUI, and at least one of them is a write operation. In such a way, we can compute all the event races in $\overrightarrow{E_b}$ and identify the K nearest event races that occur prior to the failure point as suspicious event races (shown in Line 7).

4.3 Reproducing Failures via Event Delays

Next, FlakeEcho analyzes each of the k suspicious event races identified in the previous step. For each race, FlakeEcho runs the test with an enforced order between the two racing events and verifies if the given failure is reproduced. Let e_i and e_j be the suspicious events that race on GUI data, and e_i occurred before e_j in the recorded traces. FlakeEcho introduces a delay Δ before executing e_i in the test run to enforce e_j to occur before e_i. During the execution, FlakeEcho monitors the occurrence of a failure.

To confirm whether the given failure is successfully reproduced, FlakeEcho checks if the failure in the execution is identical to the given failure. Specifically, FlakeEcho automatically compares the crash stack resulting from the execution with that of the given failure. If the crash stack of the reproduced failure matches that of the given failure, FlakeEcho confirms the given failure is successfully reproduced and outputs the delay configuration, i.e., the event to delay and the corresponding delay time.

Notably, in the current configuration, FlakeEcho flips the execution order of only one event race in each test run. This decision is grounded in the study's findings [19], which suggests that most

concurrency-related flaky tests typically involve only two threads. Therefore, for simplicity, FlakeEcho enforces order for only one event race in each test run. While enabling FlakeEcho to delay multiple event races simultaneously could theoretically increase the efficiency of reproducing flaky tests, it also raises the risk of provoking new failures owing to the compounded effect of inserted delays. However, if needed, FlakeEcho can still be configured to enforce the order for multiple event races in a test run to accommodate more complex scenarios.

5 Implementation

FlakeEcho is implemented as a fully automated flaky test reproducing framework. It extends two off-the-shelf tools: EdXposed[1] and YAMP[4] (Yet Another Methods Profiler for Android). EdXposed is a derivative of the original Xposed[3] framework, and it is used to hook the method of Android apps and to inject delays dynamically. YAMP generates a frame chart of a running test case and can monitor real-time stack traces of all threads.

6 Experiment Setup

In our experimental evaluation, we seek to answer the following research questions:

RQ1: How effective is FlakeEcho in reproducing timing-dependent flaky tests in Android apps?

RQ2: How reliably can FlakeEcho reproduce a flaky test failure?

RQ3: How does the performance of FlakeEcho change with variations in the event delay time?

RQ4: How much overhead is introduced by the dynamic analysis in FlakeEcho?

To evaluate FlakeEcho, we reproduced 80 timing-dependent flaky tests from 22 real-world Android apps. Table 1 shows the detailed information of these apps. The *Version* column specifies the version number utilized in our study, while *#LOC* measures the app's complexity, representing the lines of code written in Java and Kotlin. Additionally, the *#Stars* column indicates the app's popularity, quantified by the number of stars received on GitHub. Moreover, these applications span multiple categories, including Tools, Music & Audio, Communication, Maps & Navigation, Finance, and others, underscoring the dataset's complexity and diversity. Further test details are available in the supplementary materials.

Test Collection. To achieve diversity, we collected flaky tests from:

- **Research projects.** We reviewed recent research papers addressing flaky tests in Android applications, i.e., FlakeScanner [8], Shaker [28], and empirical studies [25, 30]. Then we manually examined flaky tests in their experiments and identified tests that are related to "asynchronous waiting" and "concurrency" issues as likely timing-dependent flaky tests. Through this process, we ended up with 47 timing-dependent flaky tests from 14 apps.
- **Github Repos.** We searched commits and bug reports on GitHub with almost 500 queries that were generated by combining three types of keywords (1) GUI-related: `gui`, `ui test`, `view`, `textview`, `button`, `widget`, `layout`, `drawable`, `theme`; (2) flaky-test-related: `flaky`, `flaky test`, `flak`, `intermittent`, `failing test`, `fix test`; (3) timing-dependency-related: `concurrent`, `concurrency`, `async wait`, `asynchronous`, `synchronization`, `child thread`, `multithread`, `timeout`. In total, we obtained 3481 related commits and 274 related bug reports. After manual examination, we deemed that 152 commits and 59 bug reports were related to timing-dependent flaky tests, and then we successfully reproduced 33 flaky test failures out of them. Finally, we obtained 33 timing-dependent flaky tests in 8 Android apps.

Table 1: Details of 22 popular open-source Android apps used in our study.

App Name	Version	#LOC	#Stars	Category
Aegis	2.0.3	43.5k	6.7k	Tools
AmazeFileManager	3.4.3	92.2k	4.8k	Tools
AntennaPod	1.8.1	102.6k	5.4k	Music & Audio
connectbot	0.4.13	122.2k	2.2k	Communication
Equate	2.0	16.0k	64	Tools
Espresso	1.0.0	17.3k	1.1k	Maps & Navigation
Feeder	1.8.9	91.6k	756	Reader
FireFoxLite	2.1.19	1598.4k	280	Communication
FlexBox	2.0.1	29.2k	18.1k	Libraries & Demo
GoogleIO	7.0.15	74.7k	21.8k	Books & Reference
Gnucash	2.4.0	90.1k	1.2k	Finance
Kaspresso	1.1.0	66.3k	1.7k	Productivity
Kiss	3.11.9	27.2k	2.7k	Personalisation
MicroPinner	2.2.0	65.0k	41	Tools
Mozilla Focus	102.0	145.3k	2.1k	Browser
MyExpenses	3.0.7.1	1835.6k	630	Finance
Omni-Notes	6.1.0	101.8k	2.6k	Note
open_flood	1.3.5	3.7k	132	Game
opt-authenticator	0.1.1	17.8k	153	Tools
Shoppinglist	2.2.1	65.2k	63	Shopping
Suntimeswidget	0.12.10	218.8k	290	Tools
YoutubeExtractor	2.0.0	2.7k	874	Video Players

Flaky Test Reproduction. To reproduce the collected flaky tests, we executed them 100 times via their test command and observed if the results contained passing and failing runs. If the results contain both passing and failing runs, we compared the stack trace of the failing runs with the developer's documentation. If they match, we consider the flaky test reproduced. Utilizing this method, we successfully reproduced 9 out of the 80 flaky tests. If a flaky test is not reproduced, we perform manual debugging using the developer's documentation to reproduce them. If the failed run shows the same behavior as in the developer's documentation, we mark the test as reproduced. With this approach, we reproduced the remaining 71 flaky tests.

Specifically, we conducted four studies to answer our research questions: (1) effectiveness study; (2) reliability study; (3) parameter sensitivity study; and (4) overhead study.

Effectiveness Study. This study aims to evaluate the effectiveness of FlakeEcho in reproducing timing-dependent flaky tests. To achieve this, we run FlakeEcho on the 80 tests to observe the number of tests FlakeEcho reproduces. In particular, we compare the stack traces of failure reproduced by FlakeEcho with the stack traces of original (initially reported) failure. If these are matched, we deem the flaky test reproduction by FlakeEcho successful.

Baseline Experiment. Additionally, we conducted a Baseline experiment to gauge how the two heuristics mentioned in Section 3

influence the efficiency of reproducing flaky tests. The baseline method reproduces a failure by exhaustively inducing a delay to any of the events that occur in the execution, delaying only one event for each run.

Moreover, we compare FlakeEcho and the state-of-the-art techniques on the 80 tests in terms of the number of reproduced flaky tests and the average time taken to reproduce failures. Unfortunately, to the best of our knowledge, there are no tools for flaky test reproduction in Android. Thus, we choose the state-of-the-art tools that *detect* flaky tests in Android, since they are closer to FlakeEcho's aim.

- Flakescanner [8], a recent work that manifests concurrency flaky test behaviors in Android apps by exploring event execution orders that may occur during test execution.
- Shaker [28], a recent technique that detects flaky tests in Android apps by introducing noise and load to the execution environment to affect event and thread execution order.

Notably, the comparison between FlakeEcho and the flaky test *detection* tools is not end-to-end. FlakeScanner and Shaker detect a flaky test by running a test to witness its flaky test failure. However, to the best of our knowledge, they are the best reference points, considering there are no other flaky failure reproduction tools for Android apps. Thus, we include them in our experiments. We use the same protocol as used in FlakeEcho to determine if a failure is successfully reproduced, i.e., if the error type and stack traces of a reported failure match with those of the given failure. In our experiment, FlakeEcho is configured with specific parameters: k is set to 4, and the delay time for an event is set to 500ms. For Flakescanner and Shaker, we followed the instructions provided in their official GitHub account. Besides, we used the default parameters given in their instructions.

Reliability Study. This study evaluates whether the delay configuration generated by FlakeEcho can reliably reproduce flaky tests. For each flaky test, FlakeEcho runs the test with a delay configuration to reproduce the flaky test failure. We run each flaky test 20 times under the provided configuration and check the number of times the flaky test failure is reproduced.

Parameter Sensitivity Study. This study evaluates the sensitivity of FlakeEcho concerning the parameter value updates. FlakeEcho can be configured with different delay values at runtime. A distinct delay value may result in a different event execution order, leading to varying effectiveness and reliability of flaky test reproduction. For this study, we use four parameters: 300ms, 500ms, and 700ms. For each parameter, we run each of the 80 tests to check the number of successfully reproduced failures. We run the test for each generated delay configuration 20 times to check how often the failure is reproduced successfully.

Overhead Study. This study investigates the overhead introduced by the dynamic analysis in FlakeEcho. As is, FlakeEcho uses dynamic tracking events and GUI access operations, which can lead to some overhead. We measure this overhead based on the test run time. Specifically, we run FlakeEcho on 80 tests to perform dynamic analysis to collect the execution time of each test run and compute the slow-down ratio by comparing the execution time of a test run without dynamic analysis.

Execution Environment. Our experiments run on a 64-bit Macbook Pro physical machine(macOS Catalina 10.15.2) with a 2.60GHz 6-Core Intel Core i7 CPU and 16GB RAM and use an Android emulator to run all test cases. The emulator is configured with 2GB RAM and the Android R operating system(SDK 11.0, API level 30). In the experiments, given a failure, only the corresponding test was executed. This setting was adopted for all the tools.

7 Experimental Results

7.1 RQ1: Effectiveness Study

In this section, we discuss FlakeEcho 's efficacy in reproducing flaky tests and compare it with FlakeScanner and Shaker. Table 2 shows the evaluation results of this study, where column "#Event" indicates the number of events generated in the test run, column "Succ" represents whether the tool successfully reproduces a flaky test failure, column "N_{succ}" denotes the number of runs out of 20 runs in which a failure is successfully reproduced with the delay configuration output by FlakeEcho, column "#Run" indicates the number of test runs performed by FlakeEcho to successfully reproduce a failure with k set to four, column "Time(s)" indicates the total execution time taken by FlakeEcho to reproduce a failure, column "Baseline" indicates the number of test runs performed by Baseline to successfully reproduce a failure, column "FS" indicates whether FlakeScanner successfully reproduces a failure, and column "SH" indicates whether Shaker successfully reproduces a failure.

Result. As shown in Table 2, FlakeEcho successfully reproduces 73 out of 80 timing-dependent flaky test failures in 22 widely used Android apps. Compared with FlakeScanner and Shaker, it achieves the best performance in terms of the number of reproduced failures, followed by FlakeScanner (40) and Shaker (27). More importantly, the results emphasize the efficiency of FlakeEcho in failure reproduction. The average time taken to reproduce a failure stands at 38.53 seconds, and as shown by "#Run", a successful reproduction only runs the test around 1.71 times for event order exploration. This efficiency is notable, considering FlakeEcho's ability to identify the faulty event order within an average of 164 events during a test run, manifesting the order in the subsequent test runs. This is because FlakeEcho employs two heuristics to optimize the event order exploration in the failure reproduction. As described in Section 4.2, FlakeEcho only considers possible event orders that originate from GUI-related event races in the test run. To further narrow down the exploration space, FlakeEcho identifies the k (k is set to four in the experiment) event races nearest to the failing point of the given failure and prioritizes exploring orders caused by these event races.

The "Baseline" column indicates the results of the baseline experiment. Specifically, the average number of runs required to reproduce a flaky test using the baseline experiments reached 51.65, marking a 30.16 times increase compared to FlakeEcho. This substantial difference highlights the effectiveness of the two heuristics utilized by FlakeEcho.

Next, we conducted an experiment to observe the entropy of runtime difference to reproduce a failure one to ten times by each tool. Since FlakeEcho can reproduce a failure deterministically, the idea is to observe the efficiency gain of FlakeEcho with an increasing number of failure reproductions compared to other techniques.

Table 2: Results of FlakeEcho, Baseline, FlakeScanner(FS), and Shaker(SH).

Id	#Event	FlakeEcho				Baseline	FS	SH
		Succ	N_{succ}	#Run	Time(s)	#Run	Succ	Succ
1	103	√	20	1	22.83	27	√	√
2	121	√	20	1	18.77	35	√	√
3	179	√	12	2	19.81	20	√	
4	145	√	19	3	196.99	71	√	
5	103	√	20	1	7.57	15	√	√
6	183	√	20	2	21.22	34		
7	33	√	20	1	15.32	9		
8	108	√	20	1	6.04	17		
9	143	√	20	1	70.16	33	√	
10	109	√	11	1	6.56	15		
11	145	√	20	3	207.01	106	√	√
12	127	√	20	2	37.37	59		
13	151	√	20	1	52.53	36	√	
14	103	√	14	1	33.87	45		
15	140	√	20	3	197.59	88		
16	223	√	20	1	6.84	64	√	
17	224	√	20	1	11.13	80	√	
18	65	√	20	1	5.43	23		√
19	57	√	20	1	8.62	20		√
20	244	√	20	3	25.36	62	√	
21	175	√	20	3	20.4	12	√	
22	83	√	20	1	6.52	35	√	
23	74	√	20	2	52.31	29	√	
24	251			4	101.32	251	√	
25	94	√	20	2	41.9	33	√	
26	86	√	20	1	14.57	20		
27	78	√	20	1	23.25	12	√	
28	34	√	11	1	1.06	6		√
29	147	√	20	1	18.67	64		
30	95			2	36.03	95	√	
31	193	√	20	1	24.58	91		
32	155			2	9.88	155		
33	152			1	8.33	152		
34	344	√	18	2	27.58	106		
35	290	√	20	1	9.95	70		
36	171	√	7	1	6.75	51		√
37	205	√	20	1	7.39	76		
38	241	√	20	1	22.7	49		
39	245	√	20	1	28.14	59		
40	208	√	20	2	88.28	80		
41	62	√	20	1	12.47	15	√	√
42	49	√	20	1	11.99	22	√	√
43	53	√	20	1	13.48	24	√	√
44	48	√	20	1	9.41	17	√	√
45	173	√	20	2	16.32	40	√	√
46	197	√	20	3	31.57	34		
47	82	√	20	1	3.58	18		√
48	109	√	20	1	23.03	45	√	√
49	396	√	20	1	122.11	52	√	√
50	284	√	20	1	135.37	30	√	√
51	162	√	13	2	13.07	16		
52	203			3	43.71	203		
53	37	√	20	1	0.85	15	√	√
54	66	√	20	1	0.55	18		
55	178			4	76.02	178		
56	212	√	20	2	10.69	17	√	
57	305	√	16	2	12.51	11		
58	282	√	18	3	35.33	29		
59	108	√	20	1	5.71	20	√	√
60	379	√	20	1	12.17	54	√	√
61	468	√	11	3	234.46	101		
62	190	√	6	4	91.84	73		
63	153	√	19	1	7.35	29	√	
64	177	√	4	1	3.44	26		
65	143	√	20	2	8.4	6		√
66	281	√	13	3	25.42	28		
67	58	√	20	2	4.25	6	√	
68	41	√	20	1	3.12	18		√
69	39	√	20	1	3.61	16	√	
70	44	√	11	2	4.96	9		
71	47	√	20	1	3.25	21		√
72	193	√	5	3	79.83	87	√	
73	135	√	20	3	19.21	65	√	
74	223	√	20	1	12.23	31	√	√
75	184	√	8	2	34.48	17		
76	49	√	20	1	2.48	11	√	√
77	273			3	79.93	273		
78	255	√	20	3	41.56	16	√	
79	407	√	13	1	35.79	49	√	√
80	323	√	11	4	276.07	87	√	√
Avg/ Sum	164	73	16.25	1.71	38.53	51.65	40	27

As is, all the tools could not reproduce all of the 80 failures. Besides, each tool also reproduces some flaky tests that others cannot reproduce. Thus, to keep the comparison fair, we compare FlakeEcho with each tool on the intersection of flaky tests that both tools could reproduce. Besides, to maximize the comparison, we compared FlakeEcho with FlakeScanner, Shaker, and 100Rerun one by one. Specifically, there were 38 test cases that both FlakeEcho and FlakeScanner could reproduce, 27 test cases that FlakeEcho and Shaker could reproduce collectively, and 9 test cases that both FlakeEcho and 100Rerun were able to reproduce. The runtime of each tool is measured from its initialization until all failures are successfully reproduced. The time taken in multiple iterations is compounded by the time taken in each iteration. Figure 6 presents the result of this experiment, and the artifacts [7] contain detailed statistics of this study. The x-axis represents the number of times a failure is reproduced, and the y-axis represents the average time to reproduce all failures by a tool.

To reproduce all failures once, FlakeScanner takes an average time of 4.07 times more than FlakeEcho, Shaker takes an average time of 18.52 times more than FlakeEcho, and 100Rerun takes an average time of 3.61 times more than FlakeEcho. It indicates that although FlakeEcho incurs the overhead of dynamic analysis, it is still the fastest on average due to its fewer number of test runs compared to the other three tools (FlakeScanner needs 7.32 runs, Shaker needs 9.58 runs, and 100Rerun needs 14.67 runs to reproduce a failure once on average). Notably, in some cases in our dataset, others outperform FlakeEcho in reproducing the first failure. For example, for cases 41, 44, and 67, FlakeScanner is faster than FlakeEcho and for cases 49, 53, and 74, 100Rerun is faster than FlakeEcho. For reproducing a failure twice, FlakeEcho is slower in only one case: Test 53 reproduction by 100Rerun. Starting from reproducing a failure thrice, FlakeEcho outperforms all. As the number of reproductions increases, the runtime advantage of FlakeEcho becomes more pronounced. It indicates that FlakeEcho achieves more success in reproducing a failure and does it more efficiently than current techniques.

We investigated the seven cases in which FlakeEcho failed to reproduce the failure. Test 24 is a test from the FireFoxLite project and involves requesting data from the Interest. The test fails when the request is not replied to in a given time, which is longer than the 500 ms FlakeEcho configured in the experiment. Thus, FlakeEcho failed to reproduce the failure. Test 30 is a test from the Kaspresso project, and its failure is not GUI data related. FlakeEcho focuses on exploring possible event orders caused by GUI data-related races and fails to reproduce this failure. Tests 32 and 33 come from the Kiss project, test 52 from the Omni-Notes project, test 55 from the Suntimeswidget project, and test 77 from the Connectbot project use a testing framework, Espresso, in which a synchronization mechanism exists for syncing the testing thread and app under test. They only fail in a corner case in which the synchronization mechanism does not cover, and the time interval for triggering these failures is extremely small. FlakeEcho failed to reproduce them. We will address those cases in further work.

In addition, it is worth noting that the discrepancy observed in FlakeScanner and Shaker's performance within our study compared to their reported results might stem from variations in the

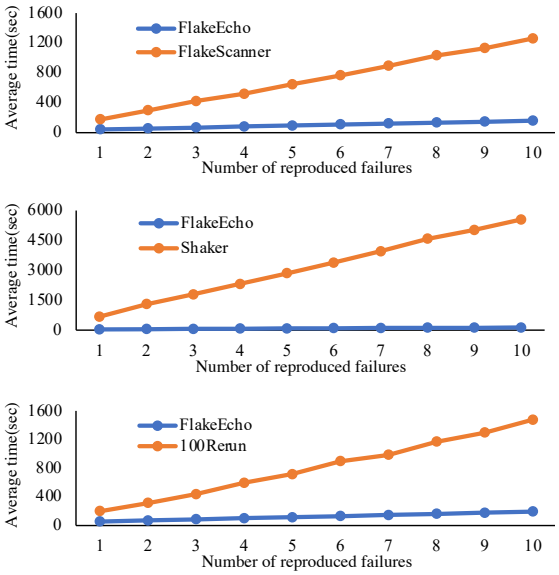


Figure 6: Average time of reproducing failures for 1-10 times by using FlakeEcho, FlakeScanner, Shaker and 100Rerun.

execution environment, encompassing differing machine configurations and the context in which the tool was applied. Additionally, the utilization of default settings may not universally guarantee optimal performance across diverse testing environments.

7.2 RQ2: Reliability Study

This study investigates how reliable is a delay configuration output by FlakeEcho in reproducing a flaky test failure. For each of the 73 successfully reproduced flaky tests, we run the test 20 times with the delay configuration output by FlakeEcho, measuring what percentage of runs can reproduce the same failure as a metric of the reliability of each flaky-test failure. This measure helps estimate how useful FlakeEcho can be for a developer aiming to repeatedly reproduce a flaky test for debugging purposes such as logging particular program states. As shown in column "N_{succ}" in Table 2, on average, the failure can be successfully reproduced for 16.25 out of 20 runs under the delay configuration output by FlakeEcho. For 53 out of the 73 reproduced flaky tests, 20 out of 20 runs can be successfully reproduced. The FlakeEcho could not reproduce the remaining cases (3.75 cases out of 20) because it did not capture the specific event that needed to be delayed; the call stack differed when the event was posted in these test runs.

In contrast, FlakeScanner schedules events to reproduce flaky tests. FlakeScanner can, in theory, provide the order of events to reproduce flaky tests repeatedly. However, in practice, it did not implement such functionality. Shaker can identify the optimal noise configuration for flaky test failure, i.e., using this noise configuration, a test is more likely to fail. However, it cannot use this configuration to reproduce a flaky test failure for every run reliably.

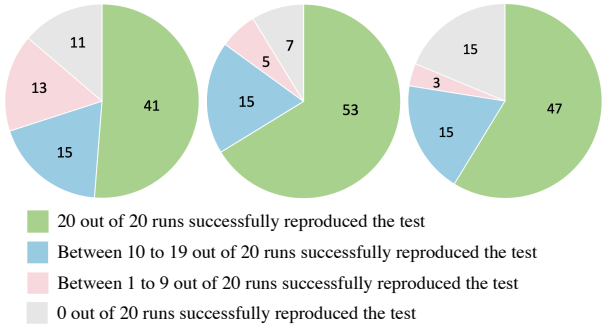


Figure 7: The number of reproduced flaky tests out of 20 runs for parameter 300ms, 500ms, and 700ms delays respectively.

7.3 RQ3: Sensitivity Study

This study investigates the sensitivity of FlakeEcho to delay values. In our experiments, we chose 300ms, 500ms, and 700ms as delay values to analyze their impact on the effectiveness and the reliability of FlakeEcho. For parameter 300ms, FlakeEcho successfully reproduced 69 out of the 80 flaky tests. For parameter 500ms, FlakeEcho successfully reproduced 73 out of the 80 flaky tests. For parameter 700ms, FlakeEcho successfully reproduced 65 out of the 80 flaky tests.

Regarding reliability, we run each of the reproduced flaky tests 20 times with the generated delay configuration for the three parameters and validate how many times a failure is successfully reproduced. The results are shown in Figure 7. FlakeEcho successfully reproduces the flaky test failure 20 out of 20 runs for 41, 53, and 47 flaky tests for parameter 300ms, 500ms, and 700ms, respectively. That is, when configured with 500ms, FlakeEcho reproduces flaky tests more reliably. According to our analysis of specific cases of experimental results, the decreased reliability associated with the 300ms delay configuration is due to an insignificant delay between the two events, i.e., more delay is needed to emit the desired flakiness. This extended gap impedes the 300ms delay configuration from effectively altering the order of events involved in the event race, leading to a diminished success rate compared to the 500ms delay configuration. On the other hand, the decreased reliability of the 700ms delay configuration arises from sporadic anomalies triggered by excessively prolonged delays (e.g., Test 12, Test 45, and Test 65). These irregularities, including sporadic application crashes, often stem from issues like prolonged UI thread waiting times, unresponsive applications, or dropped frames.

7.4 RQ4: Overhead Study

This study investigates the overhead introduced by dynamic analysis used in FlakeEcho. FlakeEcho uses dynamic analysis to record events, GUI access operations, and execute traces, which can introduce overhead in a test run. To measure the introduced overhead, we run each of the 80 tests 20 times with FlakeEcho, collect the time taken by dynamic analysis for each test, and compare it with that of a test run without any analysis. According to our experimental data, the test execution time with dynamic analysis is 1.53 times higher than the execution time without dynamic analysis on average. In

the worst case, i.e., test 26, dynamic analysis introduces three times overhead. We argue that this overhead is acceptable as flaky test reproduction (e.g., by rerunning them 100 times) is time-consuming and may take several hours if successful.

7.5 Threats to Validity and Limitations

This section discusses the potential threats to the internal and external validity of our experiments.

External Validity. Threats to external validity concern with generalizability of our evaluation results, i.e., our results may not be applicable outside of our chosen dataset. To mitigate this threat, we selected apps from the benchmarks of the related research. Besides, these tests come from 22 well-known and large apps on GitHub.

Internal Validity. Threats to internal validity concern our experimental methodology and whether it affects the outcome of our evaluation. To mitigate this, we selected flaky tests with developer documentation. Next, we validated the stack trace against the developers' documentation to validate whether a run reproduces a flaky test failure. This process is potentially error-prone due to human error. To minimize this threat, at least two researchers performed independent manual inspections and compared the results to check for discrepancies.

8 Related Work

Flaky Test Root Cause Identification. Understanding the root cause of a flaky test is an essential step toward fixing it. Ziftci and Cavalcanti [31] proposed a technique to identify the potential location of the root cause of a flaky test. Their technique relies on the execution traces of all passing and failing test runs. In particular, they expose the root cause by exploring the first point of divergence in the control flow of the failing run from any of the passing runs. Similarly, Lam et al. [14] proposed a technique to identify the root cause of flaky tests by analyzing the differences between passing and failing runs under some instrumentation. iDFlakies by Lam et al. [16] proposed an instrumentation-free approach to identify the root cause of order-dependent flaky tests. Terragni et al. [29] proposed a container-based infrastructure for identifying the root cause of flaky tests. In their approach, a flaky test is executed under various execution clusters, where each cluster explores a specific non-deterministic execution environment by fuzzing the execution environment. In contrast, our approach is instrumentation free and employs two effective heuristics to explore the space of event execution. Besides, our approach employs a simple delay injection and does not incur the overhead of rerunning a test multiple times.

Flaky Test Detection. Recent years have seen multiple approaches [5, 6, 8, 9, 16, 26, 27] to detect various types of flaky tests. Alshammari et al. [5] performed a large-scale study on flaky tests from 24 project suites to extract flaky tests' behavioral features. They developed FlakeFlagger leveraging the result of this study to identify flaky tests without rerunning them. Qin et al. [24] proposed a static approach to identify a flaky test based on the data dependency relations. Similar to FlakeFlagger, their approach also does not require rerunning the tests. Dong et al. [8] proposed an approach to detect concurrency-related flaky tests in Android via event order exploration. Bell et al. [6] employed differential analysis based on code coverage to discover unstable tests. Shi et al. [26] proposed running a test numerous times (RERUN) on each mutant to achieve consistent coverage results. This allowed them to reduce the impact of flaky tests on the mutation testing process. Dutta et al. [9] proposed a method for detecting random number-related flaky tests, i.e., tests that fail due to variations in the sequence of random numbers generated between runs. Shi et al. [27] presented a strategy for fixing order-dependent flaky tests by utilizing passing test results as a resource. In contrast, our method reliably reproduces timing-dependent flaky tests. Besides, our technique also helps the developers in debugging the reason for flakiness.

Large Scale Studies of Flaky Tests. Multiple studies [5, 10, 15, 20, 30] have explored the features and categories of Flaky tests. Many of these studies showed concurrency as one of the most prevalent causes of flaky tests. Luo et al. [20] conducted an empirical investigation of flaky tests within 51 projects. They determined concurrency to be one of the most prevalent causes of flaky testing and that the bulk of these incidents originated due to the lack of reliance on external resources. Throve et al. [30] did an empirical investigation of flaky tests in Android Applications. According to their research, more than one-third of flaky tests were caused by concurrency-related issues. Alshammari et al. [5] performed a large-scale study on flaky tests from 24 project suites to extract flaky tests' behavioral features. In this work, we developed a dataset of 80 timing-dependent flaky tests from 22 large and popular apps from GitHub.

9 Conclusion

Flaky tests pose a significant concern for all software developers since they impede regression testing and waste developer efforts. Earlier research has shown timing-dependent flaky tests as the most prominent type of flakiness. In this work, we present FlakeEcho to reliably reproduce timing-dependent flaky tests to ease the developers' debugging efforts. A timing-dependent flaky test often results from event races originating from concurrent access to GUI data. FlakeEcho employs two heuristics to prioritize exploring event orders that likely emit the event races to GUI data access. In particular, FlakeEcho prioritizes exploring the space of event orders by prioritizing the orders of events— (1) before the failure and (2) concurrently accessing GUI data. We evaluated FlakeEcho on a thoroughly collected 80 timing-dependent flaky tests obtained from 22 widely used apps. FlakeEcho successfully reproduced 73 out of 80 timing-dependent flaky tests, while the state-of-the-art techniques FlakeScanner and Shaker only reproduced 40 and 27 of them. With FlakeEcho 's efficacy in reproducing timing-dependent flaky tests, it is an ideal candidate to be utilized by developers to debug flaky tests. We make FlakeEcho and dataset publicly available to facilitate future research in this direction.

10 Data Availability

To facilitate future research on the analysis of timing-dependent flaky tests, we make FlakeEcho and our dataset available at [7].

References

[1] 2023. *EdXposed.* Retrieved 2023-8 from https://github.com/ElderDrivers/EdXposed

[2] 2023. *Flame Chart.* Retrieved 2023-8 from https://developer.android.com/studio/profile/inspect-traces

[3] 2023. *Xposed.* Retrieved 2023-8 from https://github.com/rovo89/Xposed

[4] 2023. *YAMP.* Retrieved 2023-8 from https://github.com/Grigory-Rylov/android-methods-profiler

[5] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 187–187. https://doi.org/10.1109/ICSE-Companion52605.2021.00081

[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] Xiaobao Cai, Zhen Dong, Yongjiang Wang, Abhishek Tiwari, and Xin Peng. 2024. Artifacts for Reproducing timing-dependent flaky tests in Android apps via a single delay. https://flakeecho.github.io

[8] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 367–378. https://doi.org/10.1145/3468264.3468584

[9] 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*.

[10] 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)*.

[11] Mark Harman and Peter W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. IEEE Computer Society, 1–23. https://doi.org/10.1109/SCAM.2018.00009

[12] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. *SIGPLAN Not.* 49, 6 (jun 2014), 326–336. https://doi.org/10.1145/2666356.2594330

[13] 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)*.

[14] 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*.

[15] 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*.

[16] 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*.

[17] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng (Eds.). IEEE, 403–413. https://doi.org/10.1109/ISSRE5003.2020.00045

[18] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1509–1520. https://doi.org/10.1145/3540250.3558956

[19] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 643–653. https://doi.org/10.1145/2635868.2635920

[20] 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)*.

[21] 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*.

[22] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[23] Natwar Modani, Rajeev Gupta, Guy Lohman, Tanveer Syeda-Mahmood, and Laurent Mignet. 2007. Automatically Identifying Known Software Problems. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*. 433–441. https://doi.org/10.1109/ICDEW.2007.4401026

[24] Yihao Qin, Shangwen Wang, Kui Liu, Bo Lin, Hongjun Wu, Li Li, Xiaoguang Mao, and Tegawendé Bissyandé. 2022. Peeler: Learning to Effectively Predict Flakiness without Running Tests.

[25] 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*.

[26] 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)*.

[27] 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)*.

[28] 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*.

[29] 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*.

[30] 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.

[31] Celal Ziftci and Diego Cavalcanti. 2020. De-flake your tests: Automatically locating root causes of flaky tests in code at google. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 736–745.

[32] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build? Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 113–122. https://doi.org/10.1109/ICSE-SEIP.2017.13

Received 2024-04-12; accepted 2024-07-03