An Empirical Study of Flaky Tests in Android Apps

Swapna Thorve Chandani Sreshtha Na Meng
Department of Computer Science
Virginia Tech
Blacksburg, Virginia, U.S.
{swapna6, chandani, nm8247}@vt.edu

***Abstract*—A flaky test is a test that may fail or pass for the same code under testing (CUT). Flaky tests could be harmful to developers because the non-deterministic test outcome is not reliable and developers cannot easily debug the code. A prior study characterized the root causes and fixing strategies of flaky tests by analyzing commits of 51 Apache open source projects, without analyzing any Android app. Due to the popular usage of Android devices and the multitude of interactions of Android apps with third-party software libraries, hardware, network, and users, we were curious to find if the Android apps manifested unique flakiness patterns and called for any special resolution for flaky tests as compared to the existing literature.**

For this paper, we conducted an empirical study to characterize the flaky tests in Android apps. By classifying the root causes and fixing strategies of flakiness, we aimed to investigate how our proposed characterization for flakiness in Android apps varies from prior findings, and whether there are domain-specific flakiness patterns. After mining GitHub, we found 29 Android projects containing 77 commits that were relevant to flakiness. We identified five root causes of Android apps' flakiness. We revealed three novel causes - *Dependency*, *Program Logic*, and *UI*. Five types of resolution strategies were observed to address the flaky behavior. Many of the examined commits show developers' attempt to fix flakiness by changing software implementation in various ways. However, there are still 13% commits that simply skipped or removed the flaky tests. Our observations provide useful insights for future research on flaky tests of Android apps.

***Index Terms*—Android, flaky tests, empirical**

I. Introduction

Flaky tests are the tests that terminate with non-deterministic outcomes given the same CUT. When a flaky test is executed multiple times, the testing results of some runs can be "passed" while the other runs' results are "failed". As the outcome becomes non-deterministic, developers cannot simply rely on the outcome to decide whether an app is buggy, neither can they easily debug the code because the failure symptoms may not frequently occur. Developers in the software industry have revealed the extensive existence of flaky tests [1], [2]. John Micco, a senior Google developer, once mentioned in his blog [1] that across their corpus of tests, a continual rate of about 1.5% of all test runs were seen to report a "flaky" result. Almost 16% of their tests had some level of flakiness. Unlike other bugs, flaky tests are non-deterministic and also hard to reproduce. Such tests can cost developers and testers substantial debugging time and effort.

As Android devices have become popular, the rapid development and widespread usage of Android apps require developers to heavily rely on testing for software quality assurance purposes. When test cases are flaky, developers cannot reliably test or improve their Android apps before releasing the software products. Consequently, they may dissatisfy the software consumers with unstable app functionalities.

Luo et al. conducted an empirical study on flaky tests by analyzing program commits of 51 Apache open source projects and identified the major root causes and solutions of flakiness [3]. However, they did not analyze any flaky test for Android apps. We believe that that the flaky tests in Android apps should be specially studied for three reasons:

- **Platform Fragmentation.** As the rapid evolution of the Android operating system (OS) continues, a great number of Android OS versions are available in the market, causing the problem of Android fragmentation [4]. As a result, apps are likely to behave differently across different Android platforms and manifest flaky behaviors.
- **Diverse Interaction.** Android apps usually interact with various softwares (*e.g.*, third-party libraries), hardware (*e.g.*, phone devices), networks, different screen sizes, and users. Intuitively, the more parties interacting with Android apps, the more likely that they can introduce non-determinism or uncontrollable behaviors into the execution environment, causing instability in Android apps.
- **Simple Implementation.** Compared with desktop and server applications, Android apps are usually small, and implement easy-to-understand functionalities. If we can characterize certain scenarios for flaky tests (*e.g.*, when using library A on platform B), in the future, we can build static or dynamic program analysis techniques to specially identify such scenarios in new Android apps, and provide actionable advice to address the flakiness.

Therefore, we conducted an empirical study on flaky tests of Android apps. Specifically, we crawled for Android apps on GitHub and used the keywords "flaky" and "intermittent" to search for flaky test-relevant commits. By manually examining the retrieved commits, we found 77 commits in 29 Android projects. We analyzed each commit to investigate the root cause and fixing strategy of flakiness. This study addresses the following research questions (RQs):

RQ1: What are the common causes for Android flakiness? Among the examined commits, we identified five major

causes of flakiness: Concurrency, Dependency, Program Logic, Network, and UI. Different from prior work [2], [3], our research revealed two new types of root causes: Program Logic and UI. This indicates that the flaky software behaviors are not always caused by complex thread interleaving or non-stable execution environment; instead, they may be simply due to program bugs and poor UI designs.

RQ2: What are the common strategies to fix flakiness in Android apps?

We observed five common strategies taken by developers to solve flaky tests. Four major strategies were about improving CUT, changing assertions in tests, replacing a component (e.g., library) with a more reliable one, and introducing a retry mechanism to tolerate flakiness. Nevertheless, 10 out of 77 commits showed that some developers commented out the flaky tests to remove show stoppers.

RQ3: How does flakiness in Android apps compare to flakiness in non-Android apps?

Compared with a prior study on Apache open source projects [3], our study shows that concurrency bugs are still the major cause of flaky tests. Three root causes are newly revealed for Android flakiness, such as *Dependency*, *Program Logic*, and *UI*; while some causes reported before are irrelevant to Android apps, such as test order dependency and resource leak.

II. Related Work

Some researchers have studied flaky tests and provided limited knowledge of the domain.

Vahabzadeh et.al [5] conducted an extensive quantitative and qualitative study on test bugs—the bugs that can fail a test when CUT is correct (false alarms), or pass a test when CUT is incorrect. The researchers reported semantic errors, flaky tests, and environment-related issues as the three major causes of false alarms. Luo et al. investigated 201 commits from 51 projects that fixed flaky tests [3]. They identified 10 major causes for flakiness (*e.g.*, *Async Wait* and *Concurrency*), and summarized the fixing strategies (*e.g.*, calling `waitFor()` and adding locks). However, the study did not include any flaky test of Android apps. Lin et al. recently reported that when using "AsyncTask" to write concurrent code in Android apps, developers sometimes misused the APIs and produced concurrency bugs and performance bugs [6].

As per our knowledge, this is the first empirical study on flaky tests in Android apps. Our research intends to provide valuable insights and advance the state-of-the-art knowledge of the area.

III. Methodology

A two-phase approach is followed to analyze the version-control commits: filtering and analysis.

Phase I: Filtering

We searched through GitHub to find commits relevant to the flaky tests in Android apps. This search was carried out in two steps. First, we invoked GitHub APIs [7] to search for Android projects. To compose the search query, we used keyword "Android", and names of the programming languages that are relevant to Android apps, including "C", "C#", "PhoneGap" [8], and "HTML". These language keywords effectively filtered out many irrelevant projects whose descriptions accidentally contain "Android".

In the second step, we used JGit [9] to acquire commits relevant to flaky tests from the retrieved Android projects. Specifically for each project, we downloaded the repository, enumerated all commits, and searched for keywords "flaky" and "intermittent". If a commit has its message containing either of the keywords, the commit was considered as a candidate for further analysis. To identify more flakiness-relevant commits, we also searched with other keywords like "async" and "unstable", but neither search was successful.

Phase II: Analysis

We manually inspected each candidate commit from Phase I. We filtered out a commit if:
(1) it is irrelevant to flaky tests, or
(2) it only describes the occurrence of a flaky test, but does not include any program change to fix the flakiness.

If a commit passes the filters mentioned above, we further analyzed the root cause and fixing strategy of the flaky test.

To avoid subjective bias, we did not predefine any category for root causes or fixing strategies. Instead, we took an open-coding approach [10] to reveal the categories. For every commit, we analyzed the program context, code changes, commit message, and even the related bug report(s) if available. After going through the entire dataset, we summarized the commonality between commits, tentatively defined categories based on a complete understanding of the data, and refined our categories by reexamining the entire data set several times.

Specifically, the first two authors initially checked the candidate commits from Phase I, identified the root cause and fixing strategy in each commit, and then clustered similar commits to iteratively refine categories. Next, the last author checked the categorized data and further refined the categorization. When authors had different opinions on certain commits, they were resolved by discussion and majority vote.

IV. Data Set

In Phase I, we initially found over 1000 Android projects on GitHub. With keyword-based filtering, we identified less than 50 projects that had certain commit(s) matching "flaky" or "intermittent". After further manual inspection, the dataset was narrowed down to 77 commits from 29 Android projects. Table I shows the name, category, and the number of relevant commits for each included project.

Depending on the functionalities, we classified the 29 projects into 6 categories. Especially, **Communication** includes apps for emails (*e.g.*, K-9 Mail), messaging (*e.g.*, Signal Android), and social media (*e.g.*, Monotweety). **Information Management** includes apps to manage either finance (*e.g.*,

TABLE I: Android projects under consideration

Category	Communication	Development Tool	Game	Information Management	Media	Utility
App Names	K-9 Mail **(3)**, Signal Android **(1)**, Quill **(1)**, Tusky **(1)**, Wikimedia Commons **(2)**, Wikipedia **(3)**, Monotweety **(2)**	Dagger **(1)**, FlexboxLayout **(2)**, Mute for Sonos **(1)**, PocketHub **(1)**	Dungeon Crawl Stone Soup **(5)**, Simon Tatham's Puzzles **(3)**	Walleth **(2)**, Keybase **(3)**, c:geo **(3)**	ExoPlayer **(2)**, VLC **(1)**, Voice **(2)**, AntennaPod**(1)**, NewsBlur **(1)**, MPDroid **(1)**	Realm **(10)**, StorIO **(4)**, Battery Historian **(1)**, Orfox **(1)**, AFWall+ **(1)**, ConnectBot **(2)**, OkHttp **(16)**

Walleth), security keys (*e.g.*, Keybase), or geo location information (*e.g.*, c:geo). The **Media** category includes media players (*e.g.*, ExoPlayer) and news readers (*e.g.*, NewsBlur). The **Utility** category includes apps providing utility functions like database (*e.g.*, Realm), browser (*e.g.*, Orfox), and network connection management (*e.g.*, AFWall+). In Table I, the bold numbers in parentheses (e.g., **(3)** after K-9 Mail) report the numbers of flaky test-related commits in the corresponding projects.

Our data set is smaller than the one used by prior work [3]. Two possible reasons can explain that. First, certain flaky behaviors are so difficult to debug that developers did not attempt to mention, explain, or solve them. Second, many developers do not open source their Android apps on Github (e.g. Whatsapp messenger).

V. Major Findings

We present our investigation results for the research questions separately in Section V-A, V-B, and V-C.

A. Causes of Flakiness

After analyzing the root causes of each commit, we classified commits into the six categories discussed below.

(1) Concurrency. 28 out of 77 (36%) commits have flaky tests related to concurrency bugs. These tests were flaky because developers made an incorrect assumption about the ordering of operations performed by different threads. Async wait describes the scenario in which a test makes an asynchronous call but does not wait long enough for the result to become available before using it. Prior work [3] treats async wait as a different type of root causes from concurrency, but we consider async wait to belong to concurrency. This is because when a thread is suspending, it is sometimes hard to tell whether the thread is waiting for (1) another thread to complete its task, or (2) an asynchronous call to return a value. Therefore, we put all flaky tests due to the careless design of thread interleaving into the same category.

(2) Dependency. 17 of 77 (22%) commits have flakiness caused by the usage of certain hardware, Android OS version, or a third-party library. For instance, a commit in FlexboxLayout mentioned flaky tests in some devices [11]. To avoid such device-specific flaky tests, developers relaxed the assertions to tolerate the nondeterministic values of some variables. As shown in Figure 1, the original flaky test strictly checked whether `textView1.getLeft()` and `space` had the same value. In the fixed version, developers checked whether the value of `textView1.getLeft()` was in the range [space-1, space+1].

```
…
- assertThat(textView1.getLeft(), is(space));
+ assertTrue(space -1 <= textView1.getLeft() &&
  textView1.getLeft() <= space + 1);
…
```

Fig. 1: A commit that changes assertions [11]

(3) Program Logic. 9 of 77 (12%) commits focused on fixing erroneous program logic. Developers sometimes had wrong assumptions on the apps' program behaviors, so they failed to implement functionalities properly to handle all corner cases. Figure 2 illustrates such scenarios with an example [12].

```
…
} catch (IOException e) {
  Util.closeQuietly(MockSpdyPeer.this);
-   throw new RuntimeException(e);
+   e.printStackTrace();
}…
```

Fig. 2: A commit to fix erroneous program logic [12]

In the commit, the class `MockSpdyPeer` under testing could throw an `IOException` for various reasons that are not related to the actual test. In the original version, the program caught the exception and rethrew another exception `RuntimeException`, which could terminate the app abruptly. In the fixed version, the program was modified to simply log the exception without throwing an exception. In this way, the code can avoid the intermittent app crash due to any `RuntimeException` and mitigate the overall flakiness problem.

(4) Network. 6 commits (8%) were about flakiness caused by the network. It is normal that the network can be unstable and network connections may fail now and then. However, it seems that developers initially did not think carefully about the nondeterminism, neither did they code correctly to tolerate the nondeterministic factor. Figure 3 presents such commit that fixed flakiness by tolerating the unstable network [13].

```
…
- write(IAC);
- write(SB);
+ //Use List to hold and send entire sequence at one time
+ ArrayList<Byte> byteArray = new ArrayList<>();
+ byteArray.add(IAC);
+ byteArray.add(SB);
…
```

Fig. 3: A commit that tolerates nondeterminism [13]

In the buggy version, the response sequence data was sent when `write(...)` was called for each byte in the data. As some bytes were not transferred successfully, the received

sequence was incomplete and caused errors. Therefore, to fix the bug, developers instead put the whole response sequence into an `ArrayList` object. Once all bytes are sent in a single `write(...)`, the whole sequence is either missing or received, but is never partially received by the receivers (e.g., servers).

(5) UI. 6 commits (8%) were about the flakiness of User Interfaces (UIs). When developers carelessly designed the widget layouts on UIs or misunderstood the underlying UI rendering process, some widgets were not rendered correctly or a blank screen showed up now and then. As shown in Figure 4, the original flaky test did not close the soft keyboard before invoking `click()` [14]. As a result, the clicking gesture might be wrongly captured by the soft keyboard instead of triggering the expected UI event. Developers resolved the problem by inserting `closeSoftKeyboard()` before `click()`.

```
…
- onView(withId(R.id.fab)).perform(click))
+ onView(withId(R.id.fab)).perform(closeSoftKeyboard(),
click())
…
```

Fig. 4: A commit to fix UI flakiness [14]

(6) Hard to Classify. We could not identify the root causes for 11 commits. Although developers tried to remove flakiness by replacing some code implementation or adding `if`-condition checks, it seems that developers did not understand why flakiness occurred and how their edits solved the problems. Thus, we could not infer the root causes from the flakiness description or applied fixes.

```
…
- getSherlockActivity().invalidateOptionsMenu();
+ if (getSherlockActivity() != null) {
+   getSherlockActivity().invalidateOptionsMenu();
+ }
…
```

Fig. 5: A commit to fix flakiness due to unknown cause(s) [14]

Figure 5 presents a commit attempting to fix random crash [14]. In the commit message, the developer mentioned: "*I have no idea why this crashed. Adding a guard just in case*". It seems that the developer did not understand why crashes happened intermittently. Thus, it is hard to tell how the inserted null-pointer reference check can remove the flakiness.

B. Solutions of Flakiness

From the analyzed commits, we identified five strategies developers took to reduce or eliminate flakiness. Table II presents the distribution of commits among the five strategies.

(1) Improve Implementation. As expected, developers improved software to resolve flakiness in most commits. The improvement either removed, reduced, or tolerated nondeterminism. Specifically, the flakiness issues related to *Program Logic* and *UI* were usually caused by software bugs, so developers fixed the bugs. For *Concurrency*-related issues, developers either introduced locks, added `Thread.sleep()`, or enlarged the existing thread-waiting time. By scheduling threads in a more deterministic manner, developers reduced undesirable thread interleavings. For other issues, as developers had no control over the existence of nondeterminism, they added extra condition checks or processing to make software resilient to the non-deterministic data from platforms or third-party libraries.

(2) Replace Implementation. This strategy was taken when a third-party library or network was unstable. For instance, developers replaced a library's old version with its new version to resolve flakiness [15]. Alternatively, developers could replace a code snippet with a semantically equivalent but syntactically different implementation [16], even though developers could not explain why such replacement worked.

(3) Retry. This strategy was taken when the nondeterminism was out of developers' control, *i.e.*, platform-dependent or hard to explain. For instance, when an app used Gitorious [17]—a web hosting service—to automatically download and install software modules, Gitorious might sometimes fail to grab the modules [18]. As a solution, developers implemented a retry mechanism to repeat the procedure until installation success or reaching the attempt limit (e.g., 5 trials).

(4) Modify Assertions. This solution was taken when flakiness was caused by developers' wrong assumptions on program behaviors. As shown in Figure 1, developers wrongly assumed that given the same input, the function `textView1.getLeft()` should always return the same value. Therefore, developers corrected the assertions to resolve flakiness.

(5) Remove Tests. Different from the above four strategies, this strategy is not a "real" fix, because developers skipped, commented out, or removed the flaky tests or assertions. Developers mainly took this strategy when the nondeterminism was due to dependency or unknown reasons. Although we do not believe that ignoring the flakiness is the right way to solve problems, we could sense developers' frustration when they handled these tricky scenarios [19].

C. Flakiness in Android apps vs. non-Android apps

Similar to prior work [3], we observed *Concurrency (and/or Async Wait)* and *Network* as reasons for flakiness. In addition, we revealed three new root causes for Android flakiness: *Dependency*, *Program Logic*, and *UI*. Such root causes are unique to Android apps, mainly because of (1) the variety of Android devices with different display sizes [20], and (2) any evolution inconsistency between Android OS and related third-party libraries. In the future, we can help developers resolve the flakiness due to such deterministic factors by (i) extracting bug-fixing patterns from the commits, and (ii) develop tools to automatically locate the software bugs.

As with prior work [3], we also found that developers improved their code to fix concurrency bugs and network-related issues. Additionally, we observed other meaningful strategies applied to resolve flakiness, such as replacing hardware or

TABLE II: Distribution of 77 commits among the root causes and flakiness resolution strategies

Cause \ Strategy	Improve Implementation (53)	Replace Implementation (6)	Retry (4)	Modify Assertions (4)	Remove Tests (10)
Concurrency (28)	26	-	-	2	-
Dependency (17)	4	5	3	1	4
Program Logic (9)	8	-	-	1	-
Network (6)	5	1	-	-	-
UI (6)	6	-	-	-	-
Hard to Classify (11)	4	-	1	-	6

software dependencies, replacing API invocations, introducing retry mechanisms, and updating assertions. Such mechanisms will guide us to automatically generate fixes for flakiness.

D. Threats to Validity

Our observations may not generalize well to the unexplored Android projects. Although we analyzed the commits based on our best knowledge and categorized commits by iteratively inspecting all data, our observations are still subject to human bias. The research could have been stronger if we had reproduced the flaky tests to better understand the symptoms and solutions. However, we were unable to accomplish that due to our limited domain knowledge, platform dependencies, and the inaccessibility of some test suites.

VI. Conclusion and Future Work

We are not aware of any prior work that examines Android flakiness. In this study, we identified some unique root causes and fixing strategies for the flakiness of Android apps. We also observed that different kinds of apps can have different flakiness issues. For instance, the flakiness in **Utility** apps is usually caused by *Concurrency* issues, while the flakiness in **Development Tool** and **Media** is caused by *Dependency* problems. Our study provides the following insights:

- Developers seldom describe flaky tests in commit messages with much detail, which poses challenges for other developers and researchers to analyze the problems.
- There are flakiness issues caused by deterministic factors, such as software bugs related to *Problem Logic* and *UI*. Such issues are easier to reproduce and resolve.
- Some developers simply skipped flaky tests and gave up trying to solve the problems, which indicates the difficulty of resolving flakiness and necessity of good tool support.

In the future, we plan to investigate ways to resolve flaky tests. For instance, we can define anti-patterns for any wrong usage of the APIs provided by third-party libraries, networks, or UI frameworks. We can build approaches that hardcode the anti-patterns and fixing strategies to either reduce or tolerate nondeterminism. Given a codebase with flaky tests, new approaches will generate patches, apply one patch each time, and run the patched program until all patches are investigated or the flakiness is resolved. We can also set up an online forum for developers to share their experience, and collaboratively establish the community's knowledge base of flaky tests.

Acknowledgments

We thank the reviewers for their valuable feedback.

References

[1] "Flaky tests at google and how we mitigate them," https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[2] "How to address flaky tests," https://blog.testmunk.com/how-to-address-flaky-tests/, 2018.

[3] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[4] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 83–92.

[5] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 101–110.

[6] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 341–352. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635903

[7] "GitHub API," https://developer.github.com/v3/.

[8] "PhoneGap," https://phonegap.com.

[9] "JGit," https://www.eclipse.org/jgit/.

[10] A. Strauss and J. Corbin, *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Newbury Park, California: Sage Publications, 1990.

[11] "Fix the FlexItemEditFragment not to crash when large integer is filled in the EditText," https://github.com/google/flexbox-layout/commit/1f1f57e84057b7deeddc774108a4306fbb82871d.

[12] "Prevent MockSpdyPeer interfering with Android tests," https://github.com/square/okhttp/commit/7a3bb19ea44f6771a20d3f5c323cc8bd15d73a56.

[13] "TelnetProtocolHandler Fix," https://github.com/connectbot/connectbot/commit/8e4f663.

[14] "Make sure to dismiss the keyboard to fix flaky test," https://github.com/walleth/walleth/commit/80a9ad271ac8b2a2f700b8e00236698912d1e07f.

[15] "Update ALPN," https://github.com/square/okhttp/commit/03bb6befe2867d69f631c2ef03c92b1d5a9e47e7.

[16] "MPDStatusMonitor: Workaround an Android 5 bug." https://github.com/abarisain/dmix/commit/dd8a344f6d54d8185215ed67980e4fcebb88bc55.

[17] "Gitorious," https://github.com/gitorious.

[18] "Also retry things during dependency installation," https://github.com/crawl/crawl/commit/53b308e.

[19] "Force a validation test to be ignored in the open-source maven build (for now)," https://github.com/square/dagger/commit/d7fa773181a1ff1e71a591a8583796f708d7dd21.

[20] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 308–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155604