An Empirical Analysis of UI-based Flaky Tests

Alan Romano[1], Zihe Song[2], Sampath Grandhi[2], Wei Yang[2], and Weihang Wang[1]
[1]*University at Buffalo, SUNY* [2]*University of Texas at Dallas*

Abstract—**Flaky tests have gained attention from the research community in recent years and with good reason. These tests lead to wasted time and resources, and they reduce the reliability of the test suites and build systems they affect. However, most of the existing work on flaky tests focus exclusively on traditional unit tests. This work ignores UI tests that have larger input spaces and more diverse running conditions than traditional unit tests. In addition, UI tests tend to be more complex and resource-heavy, making them unsuited for detection techniques involving rerunning test suites multiple times.**

In this paper, we perform a study on flaky UI tests. We analyze 235 flaky UI test samples found in 62 projects from both web and Android environments. We identify the common underlying root causes of flakiness in the UI tests, the strategies used to manifest the flaky behavior, and the fixing strategies used to remedy flaky UI tests. The findings made in this work can provide a foundation for the development of detection and prevention techniques for flakiness arising in UI tests.

I. Introduction

Software testing is a significant part of software development. Most developers write test suites to repeatedly test various indicators of functioning software. If a test fails, developers will analyze the corresponding test to debug and fix the software. However, not all testing results are fully reliable. Sometimes the test may show flakiness, and a test showing this behavior is denoted as a flaky test.

Flaky tests refer to tests with unstable test results. That is, the same test suite sometimes passes and sometimes fails under the exact same software code and testing code. The existence of flaky tests destroys the deterministic relationship between test results and code quality. Once a flaky test appears, it may lead to tons of efforts wasted in debugging the failed test, which leads to delays in software release cycle and reduced developer productivity [1].

In the past few years, researchers have increased efforts to address this problem [2, 3, 4]. However, the existing research on flaky tests mostly focuses on unit tests. Compared with traditional unit testing, the execution environment and automation process of UI testing are significantly different: First, many of the events in these tests, such as handling user input, making operating system (or browser) API calls, and downloading and rendering multiple resources (such as images and scripts) required by the interface, are highly asynchronous in nature. This means that user events and various other tasks will be triggered in a non-deterministic order.

Second, compared to traditional unit testing, flaky UI tests are more difficult to detect and reproduce. This is because it is difficult to cover all use-case scenarios by simulating user events to automate UI testing. UI tests introduce new sources of flakiness, either from the layer between the user and the UI or the layer between the UI and the test/application code. Moreover, the execution speed of the UI test in continuous integration environments is slow, and this difference in execution speed makes detecting and reproducing flaky tests more difficult. Therefore, researching flaky UI tests can help web and mobile UI developers by providing insights on effective detection and prevention methods.

To further investigate flaky UI tests, we collect and analyze 235 real-world flaky UI test examples found in popular web and Android mobile projects. For each flaky test example, we inspect commit descriptions, issue reports, reported causes, and changed code. We focus on the following questions and summarize our findings and implications in Table I.

RQ1: What are the typical root causes behind flaky UI tests? We examine the collected flaky UI test samples to determine the underlying causes of flaky behavior. We group similar root causes together into 4 main categories: *Async Wait*, *Environment*, *Test Runner API Issues*, and *Test Script Logic Issues*.

RQ2: What conditions do flaky UI tests manifest in and how are they reproduced? In order to understand how users report intermittent behaviors, we investigate the common strategies used to manifest the flaky UI test samples. The data reveals 5 strategies used to reproduce and report flaky UI test behavior: *Specify Problematic Platform*, *Reorder/Prune Test Suite*, *Reset Configuration Between Tests*, *Provide Code Snippet*, and *Force Environment Conditions*.

RQ3: How are these flaky UI tests typically fixed? We identify the bug fix applied to each collected flaky UI test sample and group similar ones together. We find 4 main categories for bug fixing strategies: *Delay*, *Dependency*, *Refactor Test*, and *Disable Features*.

We investigate the impacts that these UI-specific features have on flakiness, and we find several distinctions. Based on the investigation of above research questions, the main contributions of this study are:

1) Our study provides guidance for developers to create reliable and stable UI test suites, which can reduce the occurrence of flaky UI tests.
2) Our study summarizes the commonly-used manifestation and fix strategies of flaky UI tests to help developers easily reproduce and fix flaky tests, allowing them to avoid wasted development resources.
3) Our study motivates future work for automated detection and fixing techniques of UI-based flaky tests.

TABLE I: Summary of Findings and Implications

	Findings	Implications
1	Of the observed flaky tests collected, 105 tests of the 235 (45.1%) dataset are caused by an Async Wait issue.	This group represents a significant portion of the dataset collected and highlights the need to take this root cause into consideration when designing UI tests.
2	Async Wait issues are more prevalent in web projects rather than mobile projects (W 52.0 % vs M 32.5%).	The web presents an environment with less stable timing and scheduling compared with a mobile environment, so more care must be taken when network or resource loads are used within web UI tests.
3	Platform issues are happening more frequent on mobile projects rather than web projects (W 10.5 % vs M 21.7%).	It may be caused by Android fragmentation problem. So the Android developers should pay more attention to the environment configuration when choosing the test model.
4	Layout difference (cross-platform) root causes are found more in web flaky test than in mobile flaky tests (W 5.3 % vs M 1.2%).	This difference can be explained by the number of additional platform conditions that web applications can be exposed compared with the conditions found in mobile environment, such as different window sizes, different browser rendering strategies, etc...
5	Besides removing flaky test, the most common fixing strategies are refactoring logic implementations (46.0%) and fixing delays (39.3%). Among them, refactoring logic implementations can solve most issues caused by wrong test script logic, and fixing delay strategy can solve most timing issues.	Refactoring logic implementations and fixing delays should be the first-considered strategies for developers when fixing bugs.
6	Dependency fixes are more common in mobile projects than web projects (W 1.3% vs M 21.4%).	This trend can be caused by the Android fragmentation problem. Android developers should pay more attention to this problem when designing test suites.
7	Delay fixes are more common in web projects than mobile projects (W 32.2% vs M 17.9%).	This phenomenon is related to the most common test framework in Android testing, Espresso, which recommends disabling animations during tests.

TABLE II: Summary of Commit Info from UI Frameworks

UI Topic	Projects	Commits	Flaky Keyword Filtering	UI Keyword Filtering
web	999	772,901	2,553	210
angular	998	407,434	222	19
vue	998	344,526	52	1
react	997	1,110,993	603	30
svg	995	135,563	24	1
bootstrap	995	98,264	112	0
d3	980	106,160	82	1
emberjs	629	3,961	1	0
Total	7,590	2,979,802	3,649	262
Distinct	7,037	2,613,420	3,516	254

II. Background

A. Impacts of Flaky UI Tests

1) Individual Test Failures: The simplest impact that flaky test can have on test suites is that the individual test run will fail. This flaky behavior leads to a minimal amount of time and resources wasted by attempting to retry the single test.

2) Build Failures: Flaky tests that are part of continuous integration systems can lead to intermittent build failures. Flaky tests in this stage lead to wasted time trying to identify the underlying cause of the build failure only to find out that the failure was not caused by a regression in the code.

3) CI Test Timeouts: Some flaky behaviors do not cause the tests to fail outright. Instead, they cause hangups in the CI system that lead to timeouts. These hangups waste time as the system waits for a process that never finishes, causing the CI system to wait until a specified timeout is met.

III. Methodology

A. Sample Collection

1) Web: In order to collect samples of flaky UI tests, we retrieve commit samples from GitHub repositories. First, we obtain a list of the repositories leveraging popular web UI frameworks using the topic keywords 'react', 'angular', 'vue', 'emberjs', 'd3', 'svg', 'web', and 'bootstrap'. These keywords are used with the GitHub Search API [5] to identify 7,037 distinct repositories pertaining to these topics. From this set of repositories, we download all of the commits for these projects giving a total of 2,613,420 commits to search. Next, we follow a procedure similar to the one used in Luo *et al.* [2] and search the commit messages for the patterns 'flak*' and 'intermit*' and the word 'test' to find commits marked as flaky. This step reduces the number of commits to 3,516. In order to confirm which of these commits were flaky UI tests, manual inspection was performed on the commits in the list. In order to expedite the process, another keyword search is performed using the keywords 'ui', 'gui', 'visual', 'window', 'button', 'display', 'click', and 'animation' to prioritize the commits most likely to refer to flaky UI tests. This final search prioritizes 254 commit messages to search, but the full 3,516 are searched to increase the chance of identifying flaky UI test commits. After manual inspection and removing duplicate commits, the number of verified flaky tests is 152. Table II shows the summary of commit information.

2) Android: Compared with web development, Android developers are not as consistent with their choice of UI framework. Therefore, to find flaky UI tests on the Android platform, we use the GitHub Archive [6] to perform a massive search on all commits and issue reports on GitHub instead of focusing on repositories using popular UI framework. Specifically, We limit our search to the list of closed issues, since the flaky tests in closed issues are more likely to be fixed than the tests in open issues. We search with the keywords 'flak*', 'intermit*', and 'test*', which is similar to the patterns used in web searching. In order to ensure the issues we find are flaky UI tests on the Android platform, we also add constraints like 'android', 'ui', 'espresso', 'screen', etc.

B. Portion of Flaky UI Tests to Other Tests

We find that flaky UI tests collected in our methodology make up a small portion of all tests available in these repositories. This small portion can be explained by several

TABLE III: Top 10 Projects Containing the Most Flaky Tests

Project	Inspected Commits	Flaky Tests	LOC
Waterfox	937	23	3,949,098
qutebrowser	124	4	45,313
influxdb	81	12	124,591
angular	69	2	135,253
plotly.js	37	24	760,504
material-components-web	26	5	78,972
components	21	5	34,715
oppia	20	2	132,284
wix-style-react	15	11	19,830
streamlabs-obs	13	1	179,184
material-components-android	358	9	11,682
Focus-android	24	8	30,952
RxBinding	21	6	8,787
Xamarin.Forms	140	5	46,609
FirebaseUI-Android	34	4	4,386
Fenix	20	4	155,50
Detox	38	3	2,254
Components	14	3	34,715
Mapbox-navigation-android	4	2	6,201
Sunflower	3	1	354

reasons. Based on GH Archive [6], the number of open issues (over 70,000) containing potential flaky UI tests outnumbers those in closed issues (over 30,000). Open issues cannot be included in our study; however, this large number of open issues possibly containing flaky UI tests highlights the significance of UI flakiness. Besides, there are flaky UI tests not captured through the keywords. One example is in the `material-components-web` repository [2]. While our dataset is not exhaustive, we believe the results can provide a basis for future work to build on.

C. Sample Inspection

After collecting these commits and issues of flaky tests reports from GitHub, we manually inspect the collected samples to identify the information relevant to our research questions. In particular, we analyze the collected flaky tests by first inspecting the commits in the web projects and the issue reports in the Android projects for the following traits: the root cause of the flakiness, how the flakiness is manifested, how the flakiness was fixed, the test affected, the testing environment, and the lines of code of the fix. For the commits, we inspect the commit message, changed code, and linked issues. For the issue reports, we inspect the developer comments and the linked commits. When available, we also inspect the execution logs from the CI. Table III shows the information of projects containing flaky tests. Through inspection, we obtained the sample set of 235 flaky tests, of which 152 were from web repositories and 83 were from Android repositories.

D. Dataset Composition

Our dataset consists of a diverse set of flaky UI test samples. The languages of the flaky UI tests analyzed are JavaScript (63.8%), TypeScript (20.4%), HTML (8.6%), and others (7.2%) for the web projects and Java (48.2%), Kotlin (21.7%), and others (30.1%) for the Android projects.

IV. Cause of Flakiness

We investigate the collected flaky tests to determine the root cause of the flaky behavior. We manually inspect the related commits and issues of the test in order to locate the code or condition that caused the flakiness. We base our root cause categories on those defined by Luo et al. [2]. We extend the set of categories to include new categories specific to UI flakiness ("Animation Timing Issue", "DOM Selector Issue", etc...). The categorization results are summarized in Table IV.

TABLE IV: Summary of Root Cause Categories Found

Root Cause Categories	Root Cause Subcategories	Web	Mobile	Total
Async Wait	Network Resource Loading	15	4	19
	Resource Rendering	47	14	61
	Animation Timing Issue	17	9	26
Environment	Platform Issue	16	18	34
	Layout Difference	9	1	10
Test Runner API Issue	DOM Selector Issue	13	3	16
	Incorrect Test Runner Interaction	10	14	24
Test Script Logic Issue	Unordered Collections	5	0	5
	Time	1	0	1
	Incorrect Resource Load Order	11	11	22
	Test Order Dependency	6	6	12
	Randomness	2	3	5
	Total	152	83	235

A. Categorization

After manual inspection of the flaky UI tests, we identify four categories that the root causes of flakiness in these tests can fall under: (1) Timing Issue, (2) Platform Issue, (3) Test Runner API Issue, and (4) Test Script Logic Issue. We describe the categories and provide examples for each below.

1) Async Wait: We have found the root cause for a significant portion (45%) of the flaky tests analyzed arise from issues with async wait mechanisms. The common cause of such issues is that the tests do not properly schedule fetching and performing actions on program objects or UI elements, causing issues with attempting to interact with elements that have not been loaded completely. The program objects or UI elements can come from network requests, the browser rendering pipeline, or graphics pipeline. This improper ordering of events results in an invalid action and causes an exception to be thrown. Among these async wait issues, we identified three three subcategories that group similar root causes together.

a) Network Resource Loading: Flaky tests in this category attempt to manipulate data or other resources before they are fully loaded. Attempting to manipulate nonexistent data causes exceptions in the test. An example is seen in the `ring-ui` [7] web component library repository. This library provides custom-branded reusable web components to build a consistent theme across web pages. In this project, some components being tested utilize images that are fetched through network requests; however, depending on the network conditions, the images may fail to load on time or fail to load altogether. The code snippet in Figure 1 shows how the `url` variable defined in Line 1 is URL for an image network call to an external web server. The image is an avatar used by the tag component on Line 8 to display on the page. When the server call occasionally fails to respond in time due to a heavy network load, the visual test will intermittently fail as the rendered tag component will be missing the image.

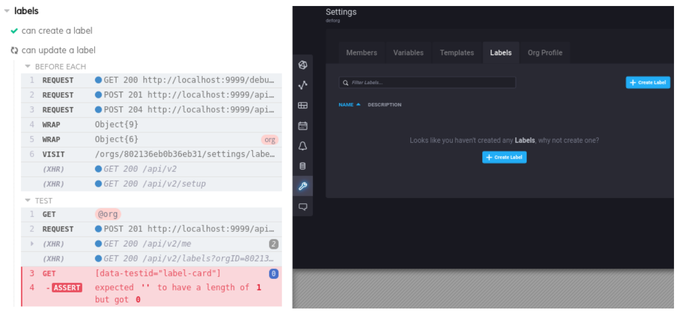
```
1  const url =
     `${hubConFigureserverUri}/api/rest/avatar/default?username=Jet%20Brains`;
2
3  class TagDemo extends React.Component {
4  render() {
5    return (
6      <div>
7        <Tag>Simple</Tag>
8        <Tag avatar={url} readOnly={false}>
9          With avatar
10       </Tag>
11     </div>
12   );
13 }}
```

Fig. 1: ring-ui Network Resource Loading Example.

Another example is found in the influxdb [8] project. This project provides a client side platform meant for storing, querying, and visualizing time series data. Figure 2 shows a flaky test for the label UI components. The test checks that labels update properly by first creating a new label and then attempting to modify the label's name and description through the UI. Figure 2a presents the view of the test suite being run on the left with the UI view on the right. Figure 2b shows the code snippet of the test corresponding to the screenshot. Lines 6-11 create the label to be used in the test. Lines 13-17 perform assertions on the labels retrieved through a network call. However, due to the execution timing, the label is not yet created in the backend store. The network call returns an empty response which causes the assertion on Line 15 to fail.

b) Resource Rendering: Flaky tests in this category attempt to perform an action on a UI component before it is fully rendered. This attempt to interact with the missing component leads to visual differences detected by screenshot tests, or exceptions thrown by attempting to access elements that have not fully loaded yet. An example of this is seen in the generator-jhipster [9] project. This project provides a platform to generate modern web application with Java. In this project, a test script attempts to click on a button and wait for the button to be displayed instead of the button being clickable. Normally, these descriptions refer to the same event, but the modal overlay shown in the UI can block the target button from being clickable. The faulty code snippet is shown in Figure 3. The waitUntilDisplayable function on Line 2 pauses the execution until the button is displayed on the page. The test can fail intermittently if another element is still above the button when Line 3 is reached, such as an element acting as a background shade in a confirmation modal.

This issue also appeared on the Android test, in the Volley [10] project, the code snippet in Figure 4 leads to flaky behavior because of a short timeout. The listener occasionally


```
labels
✓ can create a label
can update a label
  BEFORE EACH
  1 REQUEST   GET 200 http://localhost:9999/debu…
  2 REQUEST   POST 201 http://localhost:9999/api…
  3 REQUEST   POST 204 http://localhost:9999/api…
  4 WRAP      Object{9}
  5 WRAP      Object{6}                          org
  6 VISIT     /orgs/802136eb0b36eb31/settings/labe…
    (XHR)     GET 200 /api/v2
    (XHR)     GET 200 /api/v2/setup
  TEST
  1 GET       @org
  2 REQUEST   POST 201 http://localhost:9999/api…
    (XHR)     GET 200 /api/v2/me                 2
    (XHR)     GET 200 /api/v2/labels?orgID=80213…
  3 GET       [data-testid="label-card"]         0
  4 - ASSERT  expected '' to have a length of 1
              but got 0
```

(a) The test for updating a label first creates a new label through the UI. In this case, the backend had not finished processing the new label, so the network call to fetch all labels returns an empty response.

```
1  it('can update a label', () => {
2    ...
3    const newLabelName = 'attribut ()'
4    const newLabelDescription = "..."
5
6    // create label
7    cy.get<Organization>('@org').then(({id}) => {
8      cy.createLabel(oldLabelName, id, {
9        description: oldLabelDescription,
10     })
11   })
12
13   // verify name, descr, color
14   cy.getByTestID('label-card')
15     .should('have.length', 1)
16   cy.getByTestID('label-card')
17     .contains(oldLabelName)
18     .should('be.visible')
19   ...
20   // modify
21   cy.getByTestID('label-card')
22     .contains(oldLabelName).click()
23 }
```

(b) influxdb "Update Label" test code snippet.

Fig. 2: influxdb Network Resource Loading Example.

```
1  const modifiedDateSortButton =
     getModifiedDateSortButton();
2  await
     waitUntilDisplayed(modifiedDateSortButton);
3  await modifiedDateSortButton.click();
```

Fig. 3: generator-jhipster Resource Rendering Example.

does not finish executing in 100 ms timeout. This conflicts with the next request for verifying the order of calls.

```
verifyNoMoreInteractions(listener);
verify(listener, timeout(100))
        .onRequestFinished(higherPriorityReq);
verify(listener, timeout(10))
        .onRequestFinished(lowerPriorityReq);
```

Fig. 4: Volley Resource Loading Example.

c) Animation Timing Issue: Flaky tests relying on animations are sensitive to timing differences in the running environment and may be heavily optimized to skip animation events. The sensitivity to scheduling in animations can lead to issues where assertions on the events are used to test for animation progress.

An example of this type of issue is seen in the `plotly.js` project [11]. This project provides visualization components such as bar graphs, line plots, and more for use in web pages. In the transition tests, the developers find that they intermittently fail due to an underlying race condition between the call to transition the axes and the call to transition the bar graphs. Depending on which transition is called first, assertions made on the layout of the graph may fail as the bar graph elements are in different positions than expected. In Figure 5, screenshots from a code snippet provided to reproduce the different states of the animation are shown. In Figure 5a, we see that graph starts with the first bar is on value 3, the second bar is on value 4, and the third bar is on value 5. Figure 5b shows the frame immediately after the "react @ step 1" button is clicked, changing the values of the bars to 3, 6, and 5 respectively. In this figure, the background lines of the axes have been shifted in order to represent the new scale, but the bars scale incorrectly to the new axes values. Finally, Figure 5c, the bars transition to their correct new values on the new axes. Since the bars are not in the expected positions during the transition test, the assertions made fail.

Another example is seen in the `RXBinding` [12] project for Android's UI widgets. In the `RxSwipeRefreshLayoutTest`, which is used to test the swipe refresh gesture, the call to stop the refresh animation could happen anytime between the swipe release and the actual refresh animation. The behavior is flaky because the swipe animation timing used in the recorder is unable to catch up to the listener.

2) Environment: Some flaky tests manifest due to differences in the underlying platform used to run the tests. The platform can include the browser used for web projects and the version of Android, iOS, etc... used for mobile projects. We found that these issues can also be further divided into two subcategories.

a) Platform Issue: These flaky tests suffer from an underlying issue in one particular platform that causes results obtained or actions performed to differ between consecutive runs within that same platform. In the `ring-ui` project [13], the screenshot tests for a drop-down component fail due to a rendering artifact bug present on Internet Explorer. This bug causes a slight variation around the drop-down border in the screenshots taken that cause the tests to fail when compared. These tests pass when run on other browsers.

One example on Android is about `Androidx` navigation tool [14]. For some versions of Android, Espresso has flaky behavior when performing a click action on the navigated page because sometimes it cannot close the navigation drawer before the click action. However, on other versions of Android, this test always passed.

b) Layout Difference: Flaky tests can fail when the layout is different than what is expected due to differences in the browser environment. An example is found in the `retail-ui` project [15]. This project contains a set of reusable components targeted at retail sites. The screenshot test for its dropdown component fails because different default window sizes across different browsers causes the dropdown box to be cut off in some browsers.

3) Test Runner API Issue: Another root cause of flakiness we found involved an issue when interacting with the APIs provided by the testing framework that caused it to function incorrectly. Flaky tests with this root cause either use the provided APIs incorrectly, or the flaky tests manage to expose an underlying issue in the provided API that causes the functionality to differ from what was expected. We also identify two subcategories among the flaky tests observed.

a) Incorrect Test Runner Interaction: UI tests use APIs provided by the test runner to interact with UI elements, but these APIs can hit unexpected behaviors that cause incorrect behavior. For example, in the Android project `FirefoxLite` [16], flakiness appeared because the testing model registered the click action by Espresso as a long click. Figure 6 shows the UI layout after performing the click action incorrectly. A testing site should have opened by clicking "Sample Top Site" button. However, the "Remove" menu popped up instead because of the long click action on "Sample Top Site" button. This behavior difference caused the test to fail.

b) DOM Selector Issue: Flaky tests interacting with DOM elements are intermittently unable to select the correct element due to differences in browser implementations or stale elements blocking focus. An example of the flakiness arising from an incorrect DOM element selection is found in the `react-datepicker` project [17]. This project provides a reusable date picker component for the React library. The code under the test incorrectly sets two elements on the page to auto-focus on, causing a jump on the page that results in a visual flicker.

4) Test Script Logic Issue: In some flaky tests, flakiness arose due to incorrect logic within the test scripts. The flaky tests may have failed to clean data left by previous tests, made incorrect assertions during the test, loaded resources in an incorrect order, or incorrectly used a random data generator to create incompatible data. We find that tests in this category fall under one of four subcategories.

a) Incorrect Resource Load Order: Flaky tests in this category load resources after the calls that load the tests, causing the tested resources to be unavailable when the test is run. For example, in the project `mapbox-navigation-android` [18], the test crashed with an exception, because they duplicated a resource load call and then initialized a navigation activity.

b) Time: Flaky tests can fail when performing a comparison using a timestamp that may have changed from when it is created depending on the execution speed of the test. An example is found in the `react-jsonschema-form` project [19]. The project generates forms in React code by specifying the fields in JSON. In this project, a test on its date picker widget intermittently fails due to a strict comparison of time. Figure 7a shows a screenshot of a test failure

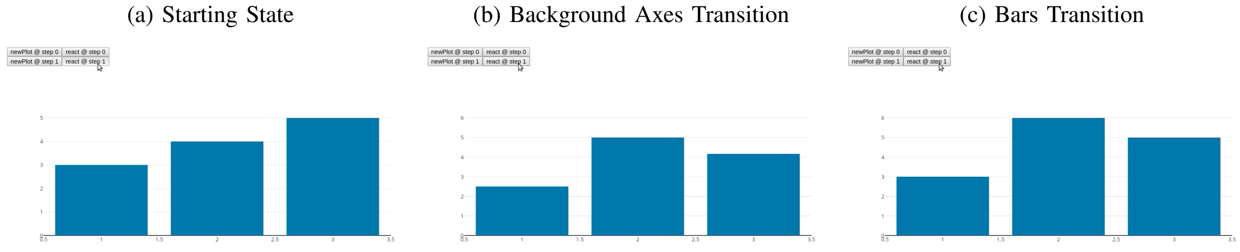


Fig. 5: An animation timing issue in the `plotly.js` project. (a) presents the initial state of a bar graph using the library. The bars start at values 3, 4, and 5, respectively. (b) The value of the bars are changed to 3, 6, and 5, respectively. The background axes change scale, but the bars are scaled incorrectly. (c) The bars then adjust to the correct scale.

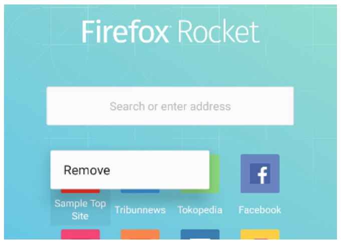
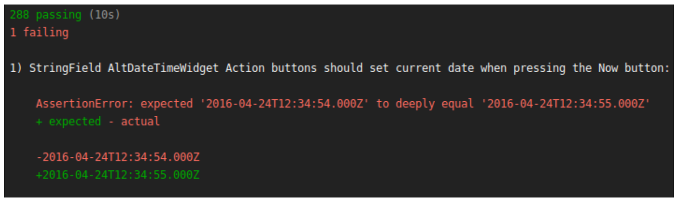


Fig. 6: `FirefoxLite` Incorrect Test Runner Interaction Example.



```
288 passing (10s)
1 failing

1) StringField AltDateTimeWidget Action buttons should set current date when pressing the Now button:

    AssertionError: expected '2016-04-24T12:34:54.000Z' to deeply equal '2016-04-24T12:34:55.000Z'
    + expected - actual

    -2016-04-24T12:34:54.000Z
    +2016-04-24T12:34:55.000Z
```

(a) CI system failure in `react-jsonschema-form` project when strictly comparing two date-time values. The values only differ by a marginal amount due to the time when the test is executed, but since the comparison is strict, the test will fail intermittently depending on when it is run.

```
1   it("should set current date when pressing the
    ↪  Now button", () => {
2    const { node, onChange } =
     ↪   createFormComponent({
3      schema: {
4        type: "string",
5        format: "date-time",
6      },
7      uiSchema,
8    });
9
10   Simulate.click(node.selector("a.btn-now"));
11   const formValue =
     ↪   onChange.lastCall.args[0].formData;
12   const expected =
     ↪   toDateString(parseDateString(new
     ↪   Date().toJSON(), true));
13   expect(comp.state.formData).eql(expected);
14  });
```

(b) `react-jsonschema-form` Strict Comparison Code snippet.

Fig. 7: `react-jsonschema-form` Strict Comparison Check Example.

within a CI system resulting from a strict comparison issue. Figure 7b presents the faulty code snippet of the flaky test that intermittently fails in the CI system. Depending on when the test is run and how quickly the statements in the test execute, the date-time value retrieved on Line 11 with the date-time value generated in Line 12 can differ by a small amount, causing the assertion on Line 13 to fail.

c) Test Order Dependency: Flaky tests in this category can interfere with or be influenced by surrounding tests in the test suit. This interference can be done through shared data stores that are not cleaned well between test runs. As a result, the data stores may contain values from previous tests and produce incorrect values as a result. One example of this is appeared in Android project `RESTMock` [20]. When trying to reset the server between tests, the test would sometimes return an exception, because there would be requests from the previous test still running as Android shares some processes between tests.

d) Randomness: Tests can use random data generation, but these tests may intermittently fail for certain values of the data generated. An example of this type of failure is found in the `nomad` project [21], which provides a platform to deploy and manage containers. In this project, they find that tests utilizing the job factory component to generate fake tasks can intermittently fail when a job given a name or URL with spaces is created. This causes encoding issues later on in the tests. Since spaces are not valid in these fields, the spaces generated by the random string generator are edge cases that should have been handled.

B. Results

From these samples, we were able to find characteristics that are particular to flaky UI tests. The most predominant root cause for these flaky UI tests involved improper handling of asynchronous waiting mechanisms, such as the mechanisms used when loading resources. These resources can include network resources as well as elements that have not yet been loaded in the page. This behavior resulted in erratic results in the tests, such as attempting to click buttons that had not yet opened. Many of these issues were resolved by refactoring the code to include delays when handling a potentially flaky call. We found that the root cause of the flaky behavior could present a challenge to find and properly fix, with some issues spanning over months to fix. In addition, the flaky nature led some of these issue reports to be closed and reopened in another report as many as five times. Other root causes included platform-specific behavior, layout differences, test order dependencies, and randomness. Platform-specific behavior produces flaky results for different runs in the same platform. Layout differences behavior causes flaky results due to inconsistencies across different platforms. Flakiness resulting from test order dependencies is caused by improper cleanup of data after runs of previous tests. UI tests involving random data generation can fail intermittently because of the characteristics of the data generated.

V. Manifestation

Reproducing flaky tests is a challenging task due to their inherit non-deterministic behavior. If developers provide details on how the flaky behavior was initially encountered and subsequently reproduced, this information provides possible strategies to apply to similar cases. We explore the strategies used by developers to manifest the underlying flaky behavior and construct categories for similar manifestations actions taken. These strategies are important when reporting the flaky test as they are inherently non-deterministic in nature, so it is challenging to reproduce them compared with regular bugs. Our categories are summarized in Table V.

TABLE V: Summary of Manifestation Categories

Manifestation Category	**Web**	**Mobile**	**Total**
Unspecified	101	40	141
Specify Problematic Platform	21	17	38
Reorder/Prune Test Suite	9	3	12
Reset Configuration Between Tests	2	7	9
Provide Code Snippet	14	6	20
Force Environment Conditions	5	10	15
Total	152	83	235

A. Specify Problematic Platform

Some tests are reported to only manifest on a specific platform. In this case, the author of the report specifies the problematic platform version to reproduce the flaky behavior. An example of this type of manifestation is found in the `waterfox` project [22]. This project is a web browser based on Firefox. In this project, an issue involving animation timing only manifests on MacOSX platforms. The report provides details on which file to run on this particular platform in order to reliably manifest the flaky behavior seen in the animation test. Another example in an Android project is from `gutenberg-mobile` [23]. This project is the mobile version for Gutenberg Editor. Figure 8 shows the bug that only appeared on the Google Pixel device with Android 10. When deleting the last character, the placeholder text should reveal as shown in Figure 8b; however, the text does not pop up. Instead, the screen appeared as shown in Figure 8a. Users would need to add an additional backspace key press to show the placeholder text.

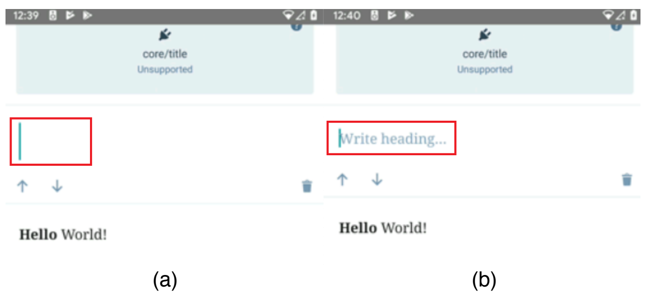


Fig. 8: `Gutenberg-mobile` Specify Problematic Platform issue Example.

B. Reorder/Prune Test Suite

Flakiness arising from test-order dependencies can be manifested by running the tests without the full test suite. This includes running tests by themselves, running tests in a different order, and changing the state between test runs in order to show the flaky behavior. An example of this manifestation strategy is seen in the `influxdb` project [24]. In this project, the flaky behavior surrounding table sorting is manifested by running the tests in the test suite independently. In some cases, trying to reset the environment configuration between tests can also lead to flakiness. In the project `RESTMock` [20], the developers tried to reduce flakiness by resetting the server configuration between tests. However, the test became more unstable because some Android processes were shared among these tests, and the forced reset caused concurrency conflicts.

C. Provide Code Snippet

Among the bug reports we observed, we find that some reports include code snippets. The code snippets extract a portion of the flaky test into an enclosed sample to make reproducing the flaky behavior more reliable. An example of this strategy is used in the project `plotly.js` project [25]. This projects provides data visualization components for use in web pages. In this project, a test for a treemap component contains a flaky image load. In order to manifest this more reliably, the reporter created a publicly-accessible code snippet that runs the component with the flaky loading behavior.

D. Force Environment Conditions

Flakiness that displays only when run on a specific platform or under certain environment settings can be manifested by forcibly setting these conditions, such as environment variables or browser window size, during the test run. An example of this can be found in the `react-datepicker` project [26]. This project provides a reusable datepicker component for use in React apps. A test for the calendar component has flaky behavior when run on the first two days of a new month. This behavior is manifested by setting the time used in the test to be one of these affected dates. Another example on Android is a click function in Espresso [27]. If we run an Espresso test which calls `openActionBarOverflowOrOptionsMenu` on a slow device, a long-click action will be accidentally performed. This bug can be manifested by a short long-click timeout.

VI. Fixing Strategy

In this section, we examine the fixes of the flaky tests. We identify common fixing patterns and group them into categories. Through comparative analysis of root causes and fixing strategies, we find that most async wait issues are fixed by increasing delay or fixing the await mechanism used. The issues caused by the environments such as platform issues and layout differences normally could not be solved. The developers prefer to fix these tests by using a workaround or changing the library version. Table VI summarizes the categories and distribution of fixing strategies and are described in the following paragraphs.

TABLE VI: Summary of Fixing Categories Found

Categories	**Subcategories**	**Web**	**Mobile**	**Total**
Delay	Add/Increase Delay	14	7	21
	Fix Await Mechanism	35	8	43
Dependency	Fix API Access	1	11	12
	Change Library Version	1	6	7
Refactor Test	Refactor Logic Implementation	49	26	75
Disable Features	Disable Animations	1	3	4
Remove Test	Remove Test	51	22	73
	Total	152	83	235

A. Delay

1) Add or Increase Delay: In order to reduce the chance of encountering flaky behavior, some tests will add or increase the delay between actions that involve fetching or loading. This prevents the rest of the test script from executing until the delay is up, giving the asynchronous call additional time to complete before moving on. An example of this fix is used in the `next.js` project [28]. This project is used to generate complete web applications with React as the frontend framework. The patch increases the delays used in multiple steps as shown in Figure 9. In the figure, Line 1 loads a new browser instance and navigates to the "about" page. Lines 2 and 3 get the text on the page and assert that it is equal to the expected value. Lines 4 and 5 manipulate the about page's component file on the filesystem to make it invalid for use. Line 6 was the delay used before of 3 seconds. If the test is run during a heavy load on the CI, the operation in Line 5 may take longer than 3 seconds, so the fix is to update the wait to 10 seconds shown in Line 7. Finally, Line 9 makes the assertion that the updated text on the page shown matched the expected error message. While this does not fix the root cause directly, this code patch does decrease the chance of running into a timing issue during testing.

```
1  const browser = await
   ↪  webdriver(context.appPort, '/hmr/about')
2  const text = await
   ↪  browser.elementByCss('p').text()
3  expect(text).toBe('This is the about page.')
4  const aboutPage = new File(join(__dirname,
   ↪  '../', 'pages', 'hmr', 'about.js'))
5  aboutPage.replace('export default', 'export
   ↪  default "not-a-page"\nexport const fn = ')
6  - await waitFor(3000)
7  + await waitFor(10000))
8  expect(await
   ↪  browser.elementByCss('body').text())
9  .toMatch(/The default export is not a React
   ↪  Component/)
```

Fig. 9: `next-js` Increase Delay Example.

2) Fix Waiting Mechanism: In order to fix flaky behavior, some tests fix the mechanisms used to wait on an asynchronous call. This ensures that the call would finish before moving forward in the test script. An example is seen in the `gestalt` project [29]. This project contains a set of reusable components used on the Pinterest website. This test is run using a headless browser, and it is accessed through the `page` variable. In Figure 10, lines 2-10 emit an event on the page to trigger the action being tested. Line 12 is supposed to pause the script execution for 200 milliseconds in order for the page to complete the action from the event handler. However, the function `page.waitFor` returns an asynchronous JavaScript promise, so it requires the `await` keyword in order to allow the promise to resolve before the lines after the call are run. The issue is fixed by adding the `await` keyword where needed.

```
1   it('removes all items', async () => {
2    await page.evaluate(() => {
3     window.dispatchEvent(
4       new CustomEvent('set-masonry-items', {
5         detail: {
6           items: [],
7         },
8       })
9     );
10   });
11
12  -  page.waitFor(200);
13  +  await page.waitFor(200);
14
15   const newItems = await
     ↪  page.$$(selectors.gridItem);
16   assert.ok(!newItems  newItems.length === 0);
17  });
```

Fig. 10: `Gestalt` Fix Waiting Example.

Another example in an Android project is from project `RXBinding` [12]. The developers avoided flakiness in this refresh layout test by manually removing the callbacks of `stopRefreshing` and adding it back after 300 ms delay, if the motion `ACTION_UP` has been caught. The code snippet is shown in Figure 11.

```
+ swipeRefreshLayout
+   .setId(R.id.swipe_refresh_layout);
+ swipeRefreshLayout.setOnTouchListener(new View.OnTouchListener() {
+   @Override public boolean onTouch(View v, MotionEvent event) {
+       if (MotionEventCompat.getActionMasked(event) == MotionEvent.ACTION_UP) {
+ handler.removeCallbacks(stopRefreshing);
+           handler.postDelayed(stopRefreshing, 300);
```

Fig. 11: `RxBinding` Fix Waiting Mechanism Example.

B. External Dependency

1) Fix Incorrect API Access: Some tests resolved the flakiness by fixing the usage of an incorrect API function. After switching this function, the test script behaved as expected. An example is shown in the `material-ui` project [30], which provides reusable web components implementing the Material design system. An API function from the testing library used to access DOM element children is incorrect. The code snippet in Figure 12 shows how the incorrect API function is fixed by calling the proper `getPopperChildren` function instead of attempting to get the element's children directly. The correct function adds additional selection criteria in order to work within the template code generated by the third-party `popper.js` [31] framework.

```
- assert.strictEqual(
-  wrapper.find(Popper)
-  .childAt(0)
-  .hasClass(classes.tooltip),
-  true
- );

+ function getPopperChildren(wrapper) {
+  return new ShallowWrapper(
+   wrapper
+    .find(Popper)
+    .props()
+    .children({ popperProps: { style: {} }, restProps: {} }),
+   null
+  );
+ }

+ const popperChildren = getPopperChildren(wrapper);
+ assert.strictEqual(
+  popperChildren.childAt(0)
+  .hasClass(classes.tooltip),
+ true);
```

Fig. 12: `material-ui` Fix Incorrect API Example.

Another example on the Android platform is found in the `Detox` project [32]. The action to launch an application in an existing instance, which has launched an app during initialization, can lead to flaky behavior. Launching an app dynamically in UIAutomator is performed by moving to the recent-apps view and then selecting the app name. However, sometimes the recent-apps view shows the full activity name (e.g. com.wix.detox.MainActivity), instead of app name (e.g. Detox), which causes flakiness. To fix this bug, developer removed the UIAUtomator API and created new instances for each launch request. Figure 13 shows the code snippet of this fixing process.

```
- device.pressRecentApps();
- UiObject recentApp = device.findObject(selector .descriptionContains(appName));
- recentApp.click();

+ final Activity activity = ActivityTestRule.getActivity();
+ final Context appContext = activity.getApplicationContext();
+ final Intent intent = new Intent(appContext, activity.getClass());
+ intent.setFlags(Intent .FLAG_ACTIVITY_SINGLE_TOP);
+ launchActivitySync(intent);
```

Fig. 13: `Detox` Fix Incorrect API Example.

2) Change Library Version: Some tests changed the version of a dependency used in the test as the developers found that the new version introduced the flaky behavior.

C. Refactor Test Checks

1) Refactor Logic Implementation: Some tests made changes to the logic used when performing checks in order to improve the intended purpose of the test while removing the flakiness observed in the test. An example is found in the `react-jsonschema-form` project [19]. In the repository, a check between consecutive timestamps is given an additional error margin to handle the case of slow execution. Figure 14 shows the code snippet changing the exact date-time comparison in Line 3 to the comparsion with an error margin of 5 seconds in Line 8.

```
1  - const expected = toDateString(
2      parseDateString(new Date().toJSON(), true));
3  - expect(comp.state.formData).eql(expected);
4  + // Test that the two DATETIMEs are within 5 seconds of each other.
5  + const now = new Date().getTime();
6  + const timeDiff = now - new Date(comp.state.formData)
7      .getTime();
8  + expect(timeDiff).to.be.at.most(5000);
```

Fig. 14: `react-jsonschema-form` Refactor Logic Implementation Example.

D. Disable Features During Testing

1) Disable Animations: In order to remove flakiness caused by animation timing, some test completely disabled animations during their run. This change removed the concern of ensuring an animation had completely finished before proceeding with the rest of the script. An example of this is seen in the the `wix-style-react` project where code is added to disable CSS animations when the test suite is run [33]. Figure 15 shows the `disableCSSAnimation` function defined on Lines 1-15 CSS rules disabling all transitions and animations. Line 21 adds a call to this function before all tests in the test suite are run.

```
1   + export const disableCSSAnimation = () => {
2   + const css = '* {' +
3   + '-webkit-transition-duration: 0s !important;'
    ↪  +
4   + 'transition-duration: 0s !important;' +
5   + '-webkit-animation-duration: 0s !important;'
    ↪  +
6   + 'animation-duration: 0s !important;' +
7   + '}',
8   + head = document.head ||
9   +
    ↪  document.getElementsByTagName('head')[0],
10  + style = document.createElement('style');
11
12  + style.type = 'text/css';
13  + style.appendChild(document.createNode(css));
14  + head.appendChild(style);
15  + };
16  ...
17  beforeAll(() => {
18    browser.get(storyUrl);
19  + browser.executeScript(disableCSSAnimation);
20  });
```

Fig. 15: `wix-style-react` Disable Animations Example.

E. Removing Tests From Test Suite

1) Remove Tests: In order to fix the test suite runs, some projects choose to remove these tests from the suite. This fix removes the flakiness in the test suite attributed to the flaky test being removed but reduces the code coverage.

2) Mark Tests as Flaky: Some tests are not entirely removed from the test suite. Instead, they are marked as being flaky which means that if the test fails, the entire test suite does not fail. This allows the test suite to be isolated from the effects of the flaky test without completely removing the coverage it provides.

3) Blacklist Tests: In order to conditionally prevent some tests from running, tests are added to a blacklist. The test in these blacklists can be skipped from test runs by setting the appropriate options for when the blacklist should be used.

VII. Discussion and Implications

We investigate our collected flaky UI tests to identify relationships between the root causes, manifestation strategies, and fixing strategies defined in Sections IV, V, and VI, respectively.

Through our inspection, we can identify relationships between the underlying root causes in issues and how the issue was fixed. These relationships are presented in Figure 16.

The goal of our study on flaky UI tests is to gain insights for designing automated flaky UI test detection and fixing approaches, so we analyze our dataset to identify correlations between manifestation strategies and root causes. However,

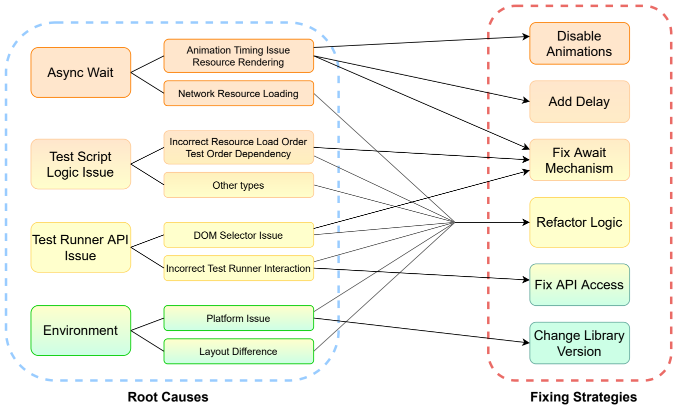


Fig. 16: Relationship Between Root Causes and Fixing Strategies.

we find that no strong correlations between these two groups exist in the dataset. Similarly, we could not identify strong correlations between manifestation strategy and fixing strategy. This leaves the question of detection strategies for flaky UI tests left open for future work to address. Our results do support relationships between root causes and fixing strategies. If the root cause of a flaky UI test is known, the relationships we draw in Figure 16 can be used to select an appropriate fixing strategy.

Preliminary design ideas can be made for some of the fixing strategies we identify in Section VI. For the *Add/Increase Delay* fixing strategy, a possible automated implementation could identify statements that perform the timing delay and increase the amount of time specified. If there is no delaying statement, then a delay can be after asynchronous function calls are performed. Granular details such as the amount of time to add in the delay or reliably identifying asynchronous function calls requires further analysis on the collected samples. Using the relationships found in Figure 16, this approach can be used to fix issues caused by *Resource Rendering* (22.4%) and *Animation Timing Issue* (15.4%). For the *Fix Await Mechanism* fixing strategy, an approach for automatic repair would be to identify statements that implement asynchronous wait mechanisms incorrectly. The details for this approach would be dependent on the language of the project and would require further analysis of the collected samples. This approach for an automated implementation of the *Fix Await Mechanism* can be used to fix *Incorrect Resource Load Order* (52.4%), *Animation Timing Issue* (38.5%), *Resource Rendering* (20.7%), and *DOM Selector Issue* (18.8%). The *Disable Animations* fix can be implemented by configuring the test environment to disable animations globally when setting up. This approach can be used to fix issues caused by *Animation Timing Issue* (7.7%), *Resource Rendering* (1.7%), and *DOM Selector Issue* (6.25%). The *Change Library Version* fixing strategy could be automated by methodically switching different versions of the dependencies used in the project. This approach could be used to address issues caused by *Animation Timing Issue* (7.7%) and *Test Runner API Issue* (6.25%).

VIII. Threats to Validity

The results of our study are subject to several threats, including the representativeness of the projects inspected, the correctness of the methodology used, and the generalizability of our observations.

Regarding the representativeness of the projects in our dataset, we focused on the most popular repositories associated with popular frameworks on web and Android. We restrict the repositories to focus on repositories that impact real applications as opposed repositories under heavy development. For mobile projects, we searched through GitHub database with strict and clear condition settings to ensure that the samples we obtain are targeted and representative.

In respect to the correctness of the methodology used, we collect all available commits on GitHub from the repositories related to popular web UI frameworks. We also leveraged the GitHub Archive repository to find all issues related to Android UI frameworks. We filter out irrelevant commits and issues using keywords and then manually inspect the remaining commits and issues in order to verify the relevance to flaky UI tests. Each sample was inspected by at least two people in order to achieve consensus on the data collected.

Regarding the generalizability of the implications made, we selected flaky test samples from actual projects used in the wild. In addition, the samples do include large-scale industrial projects, such as the Angular framework itself. We limit numeric implications only to the dataset collected, and focus on qualitative implications made on the features of the test samples.

IX. Related Work

Empirical Studies on Software Defects. There have been several prior studies analyzing the fault-related characteristics of software systems [34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]. For example, in Lu *et al.* [36] an empirical study was conducted on concurrency bugs. In Sahoo *et al.* [37], bugs in server software were studied, and in Chou *et al.* [35], operating system errors were investigated.

Studying Flaky Tests. Flaky tests have gained interest among the academic community. These tests were first looked at in 2014 by Luo *et al.* [2]. In this study, 201 commits from 51 open-source Java projects were manually inspected and categorized into 11 categories. Later, Zhang *et al.* (2014) [46] performed studied flaky tests specifically caused by test order dependencies. In 2015, Goa *et al.* [47] conducted a study that concluded that reproducing flaky tests can be difficult. Thorve *et al.* (2018) [48] studied 29 Android projects with 77 commits related to flakiness. They found three new categories differing from the ones identified in earlier studies: Dependency, Program Logic, and UI. Lam *et al.* (2019) [4] examine the presence of flaky test in large-scale industrial projects and find that flaky test cause a significant impact on build failure rates. Morán *et al.* (2019) [49] develop the FlakcLoc technique to find flaky tests in web applications by executing them under different environment conditions. Eck *et al.* (2019) [50] survey 21 professional developers from Mozilla to learn about the perceptions that developers have on the impacts that flaky tests cause during development. Dong *et al.* (2020) [51] inspect 28 popular Android apps and 245 identified flaky tests to develop their FlakeShovel technique that controls and manipulates thread execution. Lam *et al.* (2020) [52] study the lifecycle of flaky tests in large-scale projects at Microsoft by focusing on the timing between flakiness reappearance, the runtime of the tests, and the time to fix the flakiness.

Detecting and Fixing Flaky Tests. Bell *et al.* (2018) [3] developed the technique DeFlaker to detect flaky tests by monitoring the coverage of code changes in the executing build with the location that triggered the test failure. Flaky tests were those that failed without executing any of the new code changes. Lam *et al.* (2019) [4] develop the framework RootFinder to identify flaky tests and their root causes through dynamic analysis. The tool iDFlakies can detect flaky tests and classify the tests into order-dependent and non-order-dependent categories [53]. Shi *et al.* (2019) [54] develop the tool iFixFlakies to detect and automatically fix order-dependent tests by using code from other tests within a test suite to suggest a patch. Terragni *et al.* (2020) [55] proposed a technique to run flaky tests in multiple containers with different environments simultaneously.

X. Conclusions

This paper performs a study on flakiness arising in UI tests in both web and mobile projects. We investigated 235 flaky tests collected from 25 web and 37 mobile popular GitHub repositories. The flaky test samples are analyzed to identify the typical root causes of the flaky behavior, the manifestation strategies used to report and reproduce the flakiness, and the common fixing strategies applied to these tests to reduce the flaky behavior. Through our analysis, we present findings on the prevalence of certain root causes, the differences that root causes appear between web and mobile platforms, and the differences in the rates of fixing strategies applied. We believe our analysis can provide guidance towards developing effective detection and prevention techniques specifically geared towards flaky UI tests. We make our dataset available at https://ui-flaky-test.github.io/.

XI. Acknowledgments

We thank the anonymous reviewers for their constructive comments. This research was partially supported by NSF 2047980 and Facebook Testing and Verification Research Award (2019). Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] J. Micco, “The state of continuous integration testing@ google,” 2017.

[2] 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: Association for Computing Machinery, 2014, p. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[3] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[4] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 101–111.

[5] "Search," 2020. [Online]. Available: https://docs.github.com/en/rest/reference/search

[6] Github, "Github archive." [Online]. Available: https://archiveprogram.github.com/

[7] "chore: use base64 uri decoded avatar to avoid flaky ui tests if image," 2020. [Online]. Available: https://github.com/JetBrains/ring-ui/commit/f7bc28af06433ff22e898aacd2b3e8f0534defda

[8] "fix(e2e): fix race conditions," 2019. [Online]. Available: https://github.com/influxdata/influxdb/commit/4f5ff962d69a84f7a6970b02f9e79b09dbad21fe

[9] pascalgrimaud, "react: Fix intermittent e2e failures," https://github.com/jhipster/generator-jhipster/commit/2865e441e4b09335f88f3839ee9147f8b8b9c05e, 2019.

[10] sphill99 and S. Phillips, "google/volley," 2017.

[11] alexcjohnson, "one more flaky test suite," https://github.com/plotly/plotly.js/commit/a2fc07a187c4d26bf2f1bcb3e2aa806b75ad24fc, 2018.

[12] nojunpark, "Fix rxswipedismissbehavior flaky test," https://github.com/JakeWharton/RxBinding/commit/affa7a4f58e5becec4ad8b49d30f525d6ad4c2a6, 2016.

[13] princed, "Concurrent modification exception," https://github.com/JetBrains/ring-ui/commit/5d9f96d6ffa3a3c99722047677d5a545c02bdd80, 2017.

[14] chklow, "Espresso is not waiting for drawer to close," 2019. [Online]. Available: https://github.com/android/testing-samples/issues/289

[15] "test(dropdowncontainer): fix flaky screenshot test," 2019. [Online]. Available: https://github.com/skbkontur/retail-ui/commit/a006fdf0e0e65d5fde07134c6909870666e7947f

[16] cnevinc, "[ui test intermittent]," 2018. [Online]. Available: https://github.com/mozilla-tw/FirefoxLite/issues/2549

[17] Hacker0x01, "Remove second autofocus example," 2018. [Online]. Available: https://github.com/Hacker0x01/react-datepicker/pull/1390/commits/8fc31964251944be79f2e8699b79e5f39080272f

[18] Guardiola31337, "Flaky navigationvieworientationtest," https://github.com/mapbox/mapbox-navigation-android/issues/1209, 2018.

[19] "merge pull request 167 from mozilla-services/162-fix-intermittent-da," 2020. [Online]. Available: https://github.com/rjsf-team/react-jsonschema-form/commit/2318786b38ead5eddc7c0e3146825f19013e0beb

[20] Sloy, "Concurrent modification exception," 2019. [Online]. Available: https://github.com/andrzejchm/RESTMock/issues/103

[21] "Ui: Fix a couple flaky tests," 2018. [Online]. Available: https://github.com/hashicorp/nomad/pull/4167/commits/69251628f7a3f03ce603abfea5c8f48b4804c39e

[22] MrAlex94, "Bug 1504929 - start animations once after a mozreftestinvalidate," 2018. [Online]. Available: https://github.com/MrAlex94/Waterfox/commit/23793e3a2172787eca440889a8c4ec3cc6069862

[23] mchowning, "Deleting heading block content requires extra backspace to show placeholder," 2020. [Online]. Available: https://github.com/wordpress-mobile/gutenberg-mobile/issues/1663

[24] influxdata, "fix(ui): front end sorting for numeric values now being handled," 2019. [Online]. Available: https://github.com/influxdata/influxdb/commit/bba04e20b44dd0f8fd049d80f270424eb266533f

[25] etpinard, "add treemap_coffee to list of flaky image tests," 2020. [Online]. Available: https://github.com/plotly/plotly.js/commit/66156054cb08b90bc50219ff9a2baeebb674c580

[26] aij, "Fix flaky failing test," 2017. [Online]. Available: https://github.com/Hacker0x01/react-datepicker/commit/db64f070d72ff0705239f613bd5bba9602d3742f

[27] "Espresso," 2020. [Online]. Available: https://developer.android.com/training/testing/espresso

[28] vercel, "introduce dynamic(() =¿ import())," 2020. [Online]. Available: https://github.com/vercel/next.js/commit/42736c061ad0e5610522de2517c928b2b8af0ed4

[29] pinterest, "masonry: masonryinfinite for infinite fetching (307)," 2020. [Online]. Available: https://github.com/pinterest/gestalt/commit/f6c683b66b2d8b0ec87db283418459e87160a21f

[30] "[test] fix flaky popper.js test," 2020. [Online]. Available: https://github.com/mui-org/material-ui/commit/9d1c2f0ab014c76ddc042dea58a6a9384fc108f4

[31] popperjs, "Tooltip popover positioning engine," 2020. [Online]. Available: https://github.com/popperjs/popper-core

[32] d4vidi, "Fix consecutive app-launches issue," 2019. [Online]. Available: https://github.com/wix/Detox/pull/1690/commits/c982798e8904b8384e4966f4ed20700b66921b399

[33] "skip flaky visual eyes test (3306)," 2020. [Online]. Available: https://github.com/wix/wix-style-react/commit/ddebb9fc31f3aaea7b80dea034c3baa256ec2b74

[34] R. Chillarege, W. . Kao, and R. G. Condit, "Defect type and its impact on the growth curve (software development)," in *[1991 Proceedings] 13th International Conference on Software Engineering*, 1991, pp. 246–255.

[35] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," *Operating Systems Review (ACM)*, vol. 35, 09 2001.

[36] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.

[37] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 485–494. [Online]. Available: https://doi.org/10.1145/1806799.1806870

[38] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," in *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, 1992, pp. 475–484.

[39] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, pp. 271–280.

[40] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 26–36. [Online]. Available: https://doi.org/10.1145/2025113.2025121

[41] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 159–172. [Online]. Available: https://doi.org/10.1145/2043556.2043572

[42] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140. [Online]. Available: https://doi.org/10.1145/3213846.3213866

[43] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 298–308.

[44] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang, "Characterization of linux kernel behavior under errors," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, 2003, pp. 459–468.

[45] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 294–305. [Online]. Available: https://doi.org/10.1145/2931037.2931074

[46] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 385–396. [Online]. Available: https://doi.org/10.1145/2610384.2610404

[47] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 55–65.

[48] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 534–538.

[49] J. Morán, C. Augusto Alonso, A. Bertolino, C. de la Riva, and J. Tuya, "Debugging flaky tests on web applications," 01 2019, pp. 454–461.

[50] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

[51] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Concurrency-related flaky test detection in android apps," 2020.

[52] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1471–1482. [Online]. Available: https://doi.org/10.1145/3377811.3381749

[53] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, Apr. 2019, pp. 312–322, iSSN: 2159-4848.

[54] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 545–555. [Online]. Available: https://doi.org/10.1145/3338906.3338925

[55] P. S. Valerio Terragni and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," 2020.