An Empirical Study of Flaky Tests in JavaScript

Negar Hashemi*, Amjed Tahir† and Shawn Rasheed‡
Massey University
Palmerston North, New Zealand
*negar.hashemi.1@uni.massey.ac.nz, †a.tahir@massey.ac.nz, ‡s.rasheed@massey.ac.nz

***Abstract*—Flaky tests (tests with non-deterministic outcomes), can be problematic for testing efficiency and software reliability. Flaky tests in test suites can also significantly delay software releases. There have been a number of studies that attempt to quantify the impact of test flakiness in different programming languages (e.g., Java and Python) and application domains (e.g., mobile and GUI-based). In this paper, we conduct an empirical study of the state of flaky tests in JavaScript. We investigate two aspects of flaky tests in JavaScript projects: the main causes of flaky tests in these projects and common fixing strategies. By analysing 452 commits from large, top-scoring JavaScript projects from GitHub, we found that flakiness caused by concurrency-related issues (e.g., async wait, race conditions or deadlocks) is the most dominant reason for test flakiness. The other top causes of flaky tests are operating system-specific (e.g., features that work on specific OS or OS versions) and network stability (e.g., internet availability or bad socket management). In terms of how flaky tests are dealt with, the majority of those flaky tests (>80%) are fixed to eliminate flaky behaviour and developers sometimes skip, quarantine or remove flaky tests.**

***Index Terms*—Flaky Tests, Test Bugs, JavaScript**

I. Introduction

Test flakiness is a significant issue that affects the quality of software products. The impact of test flakiness is most apparent in regression testing, as regression tests rely on tests having deterministic outcomes to ensure that code changes do not break existing functionality. Flaky tests, which have non-deterministic outcomes due to factors such as concurrency, randomness, shared state and reliance on external resources, can make tests unreliable for this purpose. Such flaky behaviour is problematic as it leads to intermittent breaks in builds, uncertainty in choosing corrective measures for failing tests and bug fixes. This can have a negative impact on development, product quality, and delivery [1]–[3]. Flakiness also impacts testing-related techniques such as test suite minimization [4], test parallelization [5], as well as other techniques that rely on testing such as fault localization [6] or program repair [7].

Flaky tests are not only problematic, but they are also quite common in large codebases. For example, it was reported that around 16% of tests at Google are flaky, and 1 in 7 of the tests occasionally fail in a way that is not caused by changes to the code or tests [8], making it a real challenge for automated testing [9]. In a study of Apache projects, Vahabzadeh et al. [10] found that 21% of false alarms are caused by flaky tests. It was also shown that around 13% of failed builds in CI pipelines are due to flaky tests [11].

With the increased attention given to flaky tests in research and practice, there have been a number of studies that investigated different aspects of flaky tests, such as detection or elimination techniques. Previous studies on test flakiness and its causes largely focus on specific sources of test flakiness, such as order-dependency [12], concurrency [13] or UI-specific flakiness [14], [15]. There have been some empirical studies that investigated flakiness in the context of specific programming languages. These studies aim to understand the prevalence of test flakiness, underlying causes and strategies employed by developers to respond to them. The first study by Luo et al. [16] focusses on flakiness fixing commits mined from Apache project repositories (mostly written in Java). A similar approach was followed for Android applications in Thorve et al. [17]. A study on flakiness in Python applications [18] examines existing tests (by running the tests themselves) rather than mining commits that fix flaky tests.

To the best of our knowledge, there are no studies that specifically investigate flaky tests causes (beyond UI-causes [15]) in JavaScript projects, despite its popularity and extensive use. JavaScript has been the most commonly used programming language for several years[1]. It is also popular in various application domains, including web, mobile, and the Internet of Things (IoT) [19], [20]. In addition, given the nature of the language (handling HTTP requests, running on browsers), JavaScript is known to be one of the languages that uses asynchronous APIs extensively, with an execution model susceptible to races. Asynchronization and other concurrency bugs are known sources of flakiness in other languages, in particular Java and Python.

In this paper, we present the first extensive empirical study of flaky tests in JavaScript. In particular, the goal of this paper is to investigate how prevalent flaky tests are in JavaScript projects, and to understand the causes and impact of flaky tests in these projects. To further investigate flaky tests, we collect and analyse 452 commits from popular JavaScript projects on GitHub. For each flaky test, we inspect the commit messages, the pull requests, and the changes to the code to understand the cause of flakiness. Also, we identify the main strategies that developers follow to deal with flaky tests.

The remainder of this paper is structured as follows: we present related work on flaky tests in Section II. Our research questions and methodology are explained in Section III. We

[1]https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies

present our detailed results and answer our research questions, followed by a discussion of the results in Section IV. Finally, we present our conclusion in Section VI.

II. Related Work

Several studies published in recent years investigated the main causes and impact of flaky tests in both open-source and proprietary software. In a study of the maintenance of the regression test suite in GitHub projects [11], it was observed that about 13% of the test failures are due to flaky tests. A survey with developers on the precipitation of flaky tests by Ahmad et al. [21] identified 23 factors that are perceived to affect test flakiness (including technical and organisational factors). Similarly, Eck et al. [22] studied developers' precipitation when it comes to flaky tests (including Mozilla projects' developers) to examine the nature and the origin of 200 flaky tests that had been fixed by the same developers. It was found that flakiness is perceived as a significant issue by the vast majority of developers surveyed.

Lam et al. [23] conducted a large-scale longitudinal study of flaky tests to determine when those tests become flaky, and what changes cause them to become so. They found that 75% of the flaky tests (184 out of 245) are flaky when added, and 85% of flaky tests can be detected when detectors are run on newly added or directly modified tests. Vahabzadeh et al. [10] studied the types of bugs in test code and observed that flaky tests and semantic bugs constitute the dominant cause of tests provoking false alarms in the test code.

There have been a few empirical studies on flaky tests in different programming languages and application domains, but none we could find that studied flakiness, as an issue in itself, specifically in JavaScript. We discuss a number of those studies below, and provide a summary of flaky test causes in Table I.

Luo et al. [16] reported the first extensive empirical study of flaky tests, categorizing causes and fixing strategies of flaky tests by studying 201 commits of fixes in open-source Apache projects. The root causes of flakiness were split into ten main categories: async wait, concurrency, test order dependency, resource leak, network, time, IO, randomness, floating point operations, and unordered collections.

Gruber et al. [24] analysed projects from the Python Package Index[2] to study the cause of the detected flaky tests, and found that order dependency is the main cause of flakiness in Python projects. The study identified *infrastructure flakiness* as a new type of test flakiness that has not been reported previously. Lam et al. [25] studied the lifecycle of flaky tests in six large-scale projects at Microsoft. They found that asynchronous calls are the leading cause of flaky tests in those Microsoft projects. They also proposed an automated solution, called Flakiness and Time Balancer (FaTB), to reduce the frequency of flaky-test failures caused by asynchronous calls.

The impact of flaky tests has also been studied in different application domains. A study of flaky tests in Android apps [17] identified two new causes to those discussed in [16], namely: program logic and UI. Dutta et al. [26] studied flaky tests in applications that use probabilistic programming or machine learning frameworks, finding that *randomness*, as expected, is the biggest cause of flakiness in those applications. Romano et al. [15] analysed 235 flaky UI tests from 62 web and Android projects, and identified four categories of causes for UI-based flaky tests. Moran et al. [27] focused on detecting the root cause of flakiness in web applications by analysing test execution under different combinations of the environmental factors that may trigger flakiness.

There have also been a number of tools that have been targeted to detect certain types of flaky tests such as *DeFlaker* [28], *RootFinder* [29], *iFixFlakies* [30], *SHAKER* [31] and *FlakeScanner* [32].

III. Study Methodology

The main goal of this study is to investigate causes and strategies followed to deal with flaky tests in JavaScript. In this paper we answer the following two research questions:

RQ1 What are the main causes of flaky tests in JavaScript projects?

RQ2 What are the strategies followed when dealing with flaky tests in JavaScript projects?

Our target is to analyse commits that are believed to refer to tests with flaky behaviour, rather than exposing flakiness by compiling and analysing the programs themselves. We conducted the study in two phases. We first collect commits (including commit messages and source code changes) from open-source JavaScript projects and then manually analyse the causes of flakiness as noted in these commits. We also analyse the strategies followed when those flaky tests were fixed.

A. Dataset

We constructed a dataset of JavaScript projects obtained from GitHub. We targeted only popular projects based on the number of stars of each project (based on the *star* rating on GitHub). GitHub's star feature allows users to mark their interest or helpful repositories on GitHub. In a way, it is a metric that shows how much interest a project has drawn. We found that the number of stars is highly correlated with the number of forks and contributors, indicating popularity.

To search through the repositories, we used a GitHub API[3] to search through all publicly available GitHub repositories and filter results of the commits for analysis. We first extract the top 40 starred JavaScript projects on GitHub. We choose the top 40 projects because they provide a large enough sample (number of commits) that is deemed suitable for our analysis. We compared the number of commits we retrieved by the ones in similar previous studies: Luo et al. [16] retrieved 486 commits in total, but analysed 201 commits that fix flaky tests; while Thorve et al. [17] retrieved 77 commits. Romano et al.

[2]https://pypi.org/

[3]https://docs.github.com/en/rest/reference/search

TABLE I
CAUSES OF FLAKY TESTS AS IDENTIFIED IN PRIOR STUDIES

Luo et al. [16]	**Thorve et al.** [17]	**Dutta et al.** [26]	**Gruber et al.** [24]
Async Wait	Concurrency	Algorithmic Non-determinism	Infrastructure
Concurrency	Dependency	Floating-point Computations	Test Order Dependency
Test Order Dependency	Program Logic	Incorrect/Flaky API Usage	Network
Resource Leak	Network	Unsynced Seeds	Randomness
Network	UI	Concurrency	IO
Time		Hardware	Time
IO		Other	Async Wait
Randomness			Concurrency
Floating Point Operations			Resource leak
Unordered Collections			Test Case Timeout
			Unordered Collections

[15] analysed 235 flaky UI test. In comparison, we extracted a total of 316,948 commits and ended up with 735 flaky tests' related commits.

B. Mining Process

Our mining process is shown in Fig. 1. By using GitHub API, we searched through 40 repositories for the following keywords: "flaky", "flakey", "flakiness", "intermit", "fragile", "brittle", "Intermittent", and "non-deterministic test". We found 18 repositories to have at least one commit with one of the above keywords we used. This has resulted in retrieving a total of 735 commits.We conducted this search on May 8th, 2021. Table II shows statistics from the projects we analysed.

Out of those 735 commits, we found that 254 of them are related to languages other than JavaScript, so we excluded those as we are interested only in flakiness that is related to JavaScript. This left us with a total of 481 commits that are relevant to our analysis.

C. Manual Analysis of Commits

We manually analysed all 481 commits that we included. For each commit, we analysed the commit message, any code changes in the files in the repository as well as the associated issues (if any) in the issue tracker. We started our process by manually filtering out commits related to languages other than JavaScript (out of 735 commits, we found 481 to be JavaScript-related). The first two authors then individually and separately classified all 481 commits based on the cause of flakiness and then compared their classification results. To inform our classification, we checked the associated issues (if any), pull requests and changes in the test and CUT. When there were classification disagreements between the two authors, the third author was then involved to resolve conflicts. The third author analysed a total of 139 commits (where the first two authors did not agree or were not sure). Disagreements between all authors were settled through discussions until a 100% agreement was reached.

We categorize each of these commits into one of the following :

- **Relevant**: commits that are directly related to flaky tests.
- **Irrelevant**: commits that contain one of the keywords, but it is not related to flakiness.
- **Duplicated**: identical commits.

Of those 481 commits, we found that 11 commits were found to be irrelevant (not related to test flakiness), and 18 commits were duplicated, thus those were excluded. This resulted in a total of 452 commits that we considered relevant, which we manually classified. We provide a full replication package (including scripts and data) online at: https://doi.org/10.5281/zenodo.6757825.

IV. RESULTS

A. RQ1: What are the main causes of flaky tests in JavaScript projects?

To answer RQ1, we manually analysed and categorised all commits based on the relevant causes of flakiness (based on the commit message or the change in the files associated with the commit). We used the list of causes noted in previous empirical studies on test flakiness as our baseline (See Table I). We then classified each commit based on the list of flakiness causes. If the cause of flakiness is new or not in the list, we then add it as a new cause and continue with our classification.

During this process, we were not able to categorise or determine the cause of flakiness in 94 commits. Hence, those were categorised as *"unsure"* or *"hard to categorise"* and we continued to analyse the remaining 358 commits.

As it turns out, most of the causes we identified fit well into one or more of the categories noted in previous studies (i.e., [16], [17], [24], [26]). However, some of the causes we identified resulted in new categories that best reflect the nature of flakiness and thus those have been categorised separately.

Table III shows the results of the top 10 causes we found in the JavaScript projects we analysed. We present the results of our categories below, together with examples of each cause from the projects we analysed.

Concurrency We classify a commit in this category when a test is flaky due to any concurrency-related issues in the test or code under test (CUT) such as event races, atomicity violations or deadlocks [33]. We found that 74 of the flaky tests belong

TABLE II
PROJECTS FROM THE TOP 40 MOST STARRED JAVASCRIPT PROJECTS ON GITHUB WITH COMMIT MESSAGES CONTAINING AT LEAST ONE OF THE SEARCH KEYWORDS

Repositories	#Stars	#Commits	JavaScript File	#Flakiness Related Commits	#Contributor	Age	Description
freeCodeCamp/freeCodeCamp	320k	28253	47.4%	4	4287	Oct. 2014	Corpus of educational JavaScript projects
vuejs/vue	188k	3200	97.6%	2	388	Dec. 2013	JavaScript framework for building UI
facebook/react	174k	14448	95.7%	22	1495	July 2013	JavaScript library for building user interfaces
twbs/bootstrap	153k	21165	41.4%	5	1231	Oct. 2011	Web development JavaScript framework
electron/electron	96.6k	25574	6.5%	48	1069	July 2013	Desktop apps framework
nodejs/node	81.6k	34487	61.2%	467	3012	May 2009	Cross-platform JavaScript runtime environment
denoland/deno	77.7k	6310	23.6%	3	635	May 2018	A JavaScript and TypeScript runtime environment
mrdoob/three.js	74.6k	38172	54.2%	1	1482	Apr. 2010	JavaScript 3D Library
mui-org/material-ui	70.9k	17626	58.2%	19	2278	Nov. 2014	A framework to build React apps
storybookjs/storybook	64.5k	35707	9%	14	1365	April 2016	UI components development tool
atom/atom	56k	38404	88.3%	68	488	Dec. 2013	Cross-platform text editor
jquery/jquery	55.3k	6536	93.6%	9	276	Jan. 2006	JavaScript Library
chartjs/Chart.js	54.7k	4043	98.3%	1	380	June 2014	Data visualization library
ElemeFE/element	50.8k	4500	26.7%	1	556	Sep. 2016	A Vue.js-based UI Toolkit for Web
lodash/lodash	50.6k	8005	100%	2	310	April 2012	A modern JavaScript utility library
moment/moment	46k	3961	99.7%	8	590	Jan. 2015	A JavaScript date library
meteor/meteor	42.6k	24211	92.3%	50	465	Jan. 2012	JavaScript web framework
yarnpkg/yarn	40.1k	2346	98.7%	11	521	June 2016	JavaScript package manager
Total		316948		735	20828		

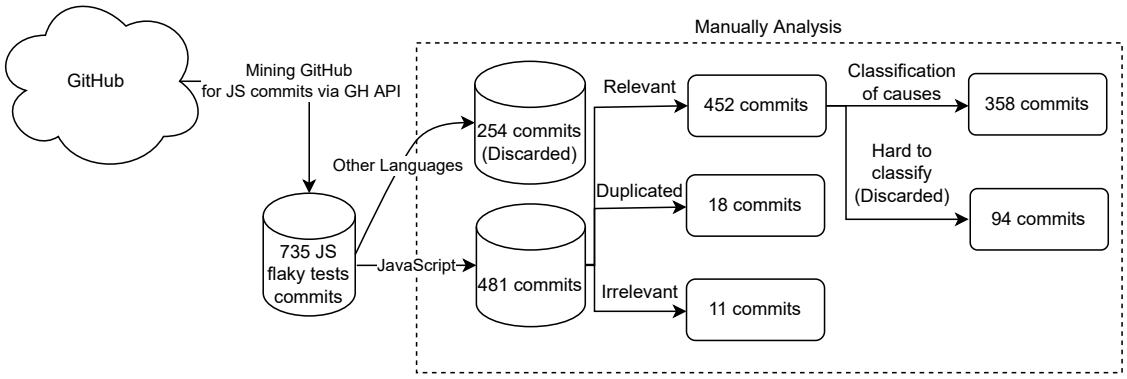

Fig. 1. Mining flaky tests' commits from GitHub's JavaScript projects

to this category. For example, in this snippet[4] from Node.js, using `setImmediate` in line 6 could lead to race conditions, which lead to non-deterministic test outcomes.

```
1 onst keepOpen = setTimeout(() => {
2 @@ -20,7 +20,7 @@ const timer = setInterval(()↩
      => {
3     timer._onTimeout = () => {
4       throw new Error('Unrefd interval fired ↩
          after being cleared.');
5     };
6     setImmediate(() => clearTimeout(keepOpen));
7   }
8 }, 1);
```

Another example of a concurrency related flaky tests is the following example from Meteor[5]. This test is flaky because the template rendered callbacks get called after flush time, but not if the template got destroyed. If the client managed to respond to the server rejecting the method before the client's flush cycle, the rendered callback would never fire.

```
1 testAsyncMulti('spacebars - template - defer ↩
    in rendered callbacks', [function (test, ↩
    expect) {
```

[4]https://tinyurl.com/2s32r7xk

[5]https://tinyurl.com/ysde5t37

TABLE III
OVERALL RESULTS OF FLAKY TEST CATEGORIES (CAUSES) IN JAVASCRIPT

Cause of Flakiness	# of Commits	%
Concurrency	74	20.7%
Async Wait	70	19.6%
OS	66	18.4%
Network	45	12.6%
Platform	37	10.3%
UI	21	5.9%
Hardware	17	4.7%
Time	12	3.4%
Resource Leak	10	2.8%
Other	13	3.6%
Total	365*	

* The total number of commits is 358 as there are 7 commits related to 2 causes of flakiness.

```
  var tmpl = Template.spacebars_template_test_defer_in_rendered;
  var coll = new Meteor.Collection("test-defer-in-rendered--client-only");
  tmpl.items = function () {
    return coll.find();
  };
  var subtmpl = Template.spacebars_template_test_defer_in_rendered_subtemplate;
  subtmpl.rendered = expect(function () {
    Meteor.defer(function () {
    });
  });
  var div = renderToDiv(tmpl);
  Meteor._suppress_log(1);
  coll.insert({});
}]);
```

Async Wait A commit is labelled to be in this category when a test makes an asynchronous call and does not wait properly for the result of the call to become available before using it. Async Wait is in fact a subcategory of concurrency, but we classify it separately here mainly because it represents nearly half(48%) of concurrency-related flaky tests. In the projects we analysed, there are 70 commits in total that fits into the async wait category, such as the following example from Electron[6]:

```
1  it('should clear the navigation history', async () => {
2    loadWebView(webview, {
3    nodeintegration: 'on',
4    src: `file://${fixtures}/pages/history.html`
5  })
6  const event = await waitForEvent(webview, 'ipc-message')
```

In this test, the IPC message was sent before it was fully loaded, so the history did not contain everything it should have, and cannot clear all the pages, leading to non-deterministic outcomes.

[6]https://tinyurl.com/2p8zr5k4

In the following test[7] from FreeCodeCamp, using `cy.contains('Go to next challenge').click();` in line 10 and then not waiting for the result before visiting the next page can cause flakiness.

```
1  describe('project submission', () => {
2    it('Should be possible to submit Python projects', () => {
3     const { superBlock, block, challenges } = projects;
4     challenges.forEach(challenge => {
5       cy.visit(`/learn/${superBlock}/${block}/${challenge}`);
6       cy.get('#dynamic-front-end-form')
7         .get('#solution')
8         .type('https://repl.it/@camperbot/python-project#main.py');
9       cy.contains("I've completed this challenge").click();
10      cy.contains('Go to next challenge').click();
```

Operating Systems (OS) A commit is classified to be in the OS category when the test fails due to a run in a specific OS or OS version, while passing on others. It can happen because of a test dependency on specific OS features or environmental settings. We classify OS as a separate category of root causes from the platform (discussed later) because it represents a large percentage of flaky tests by itself. In total, we found that 66 commits belong to the OS category. The following test[8] from Node.js is a good example of a flaky test in this category.

```
1  if (process.platform === 'darwin') {
2    setTimeout(function() {
3      fs.writeFileSync(filepathOne, 'world');
4    }, 100);
5   } else {
6    fs.writeFileSync(filepathOne, 'world');
7  }
```

In OS X, events for `fs.watch()` might only start showing up after a delay. To work around that, there is a timer in the test that delays the writing of the file by 100ms, but in some cases this might not be enough. So the event never fired, and the test times out.

Another example of an OS flaky test from Angular[9] shows a test that resulted in a `ECONNREFUSED` error (i.e., refused connection) for only Windows, but it works well for other OS.

```
1  function launchChromeAndRunLighthouse(url, flags, config) {
2    const launcher = new ChromeLauncher({autoSelectChrome: true});
3    return launcher.run().
4      then(() => lighthouse(url, flags, config)).
5      then(results => launcher.kill().then(() => results)).
6      catch(err => launcher.kill().then(() => { throw err; }, () => { throw err; }));
7  }
```

Network A commit is in this category when a test fails due to remote connection failures (e.g., lost internet connection when

[7]https://tinyurl.com/2p8ebjbx
[8]https://tinyurl.com/533ycmsc
[9]https://tinyurl.com/2p9d7rxk

accessing an external URL) or local bad socket management. There are 45 commits that fit into the network category. One example we see from this category is the following test from Node.js[10], which is using port '0' for an IPv6-only operation and assuming that the OS would supply a port that was also available in IPv4. However, the CI results seem to indicate that a port can be supplied that is in use by IPv4 but available to IPv6, resulting in the test failing.

```
net.createServer().listen({
  host,
  port: 0,
  ipv6Only: true,
}, common.mustCall());
```

Another example from React[11] shows a test that uses absolute URLs, which can end up in flakiness.

```
var __dirname = __filename.split('/').reverse().slice(1).reverse().join('/');
window.ReactWebWorker_URL = __dirname + '/../src/test/worker.js' + cacheBust;
document.write('<script src="' + __dirname + '/../build/jasmine.js' + cacheBust + '"><\/script>');
document.write('<script src="' + __dirname + '/../build/react.js' + cacheBust + '"><\/script>');
document.write('<script src="' + __dirname + '/../build/react-test.js' + cacheBust + '"><\/script>');
document.write('<script src="' + __dirname + '/../node_modules/jasmine-tapreporter/src/tapreporter.js' + cacheBust + '"><\/script>');
document.write('<script src="' + __dirname + '/../test/the-files-to-test.generated.js' + cacheBust + '"><\/script>');
document.write('<script src="' + __dirname + '/../test/jasmine-execute.js' + cacheBust + '"><\/script>');
```

Platform The commits in this category include all flaky tests related to platform, e.g., test environment, browsers. We excluded OS related flaky tests here as those, although are platform related, are a large enough subcategory that we discussed separately earlier in this section. We identified 37 commits in total from the platform category. The following example[12] from Electon shows a test that, due to some specifications, failed on one CI, but passed on another. The test times out regularly on Travis CI but pass when built on Jenkins.

```
it('can be manually resized with setSize even when attribute is present', done => {
  w = new BrowserWindow({show: false, width: 200, height: 200})
  w.loadURL('file://' + fixtures + '/pages/webview-no-guest-resize.html')
```

Similarly, the following test[13] from Node.js is flaky due to reliance on platform timing in lines 3, 5, 8, 9, 11, and 18.

```
1  const common = require('../common');
2  const fs = require('fs');
3  const platformTimeout = common.platformTimeout;
4  const t1 = setInterval(() => {
5    common.busyLoop(platformTimeout(12));
6  }, platformTimeout(10));
7  const t2 = setInterval(() => {
8    common.busyLoop(platformTimeout(15));
9  }, platformTimeout(10));
10 const t3 =
11   setTimeout(common.mustNotCall('eventloop blocked!'), platformTimeout(200));
12 setTimeout(function() {
13   fs.stat('/dev/nonexistent', () => {
14     clearInterval(t1);
15     clearInterval(t2);
16     clearTimeout(t3);
17   });
18 }, platformTimeout(50));
```

UI The commits in this category include all flaky tests related to UI features, e.g., flakiness due to different windows display, the blinking cursor, or the highlight decoration. There are 21 tests that belong to the UI category. The following example[14] from Atom shows a test aims to verify that the cursor blinks when the editor is focused and the cursors are not moving. It expects the blinking cursor to start in the visible state, and then transition to the invisible state. But the initial state of the cursor is not important, since the cursor toggles between the visible and the invisible state.

```
1  await component.getNextUpdatePromise()
2      const [cursor1, cursor2] = element.querySelectorAll('.cursor')
3      expect(getComputedStyle(cursor1).opacity).toBe('1')
4      expect(getComputedStyle(cursor2).opacity).toBe('1')
5      await conditionPromise(() =>
6        getComputedStyle(cursor1).opacity === '0' && getComputedStyle(cursor2).opacity === '0'
7      )
8      await conditionPromise(() =>
9        getComputedStyle(cursor1).opacity === '1' && getComputedStyle(cursor2).opacity === '1'
10     )
11     await conditionPromise(() =>
12       getComputedStyle(cursor1).opacity === '0' && getComputedStyle(cursor2).opacity === '0'
13     )
```

Also, the following Atom test[15] is flaky due to UI related issues. Resize events are unreliable and may not be emitted right away. This could cause the test code to wait for an update promise that was unrelated to the resize event (e.g., cursor blinking).

```
1  setEditorHeightInLines(component, 13);
2  await setEditorWidthInCharacters(component, 50);
3  expect(component.getRenderedStartRow()).toBe(0);
```

[10]https://tinyurl.com/yc6c97m6
[11]https://tinyurl.com/yfprp6tx
[12]https://tinyurl.com/3xms6muw
[13]https://tinyurl.com/348bzef5
[14]https://tinyurl.com/yckkhh4a
[15]https://tinyurl.com/mvdvm9vf

```
4   expect(component.getRenderedEndRow()).toBe(13)↩
        ;
```

Hardware This category combines all flakiness in test outcomes that are resulted from hardware-dependencies. We found a total of 17 commits related to the hardware category. For example, in the following example[16] from Node.js, the test runs in Raspberry Pi. The number of clients is set to 100, which has caused some flaky behaviour when run in Raspberry Pi.

```
1  var responses = 0;
2  var N = 10;
3  var M = 10;
4  server.listen(common.PORT, function() {
5   for (var i = 0; i < N; i++) {
6    setTimeout(function() {
7     for (var j = 0; j < M; j++) {
8      http.get({ port: common.PORT, path: '/' ↩
           }, function(res) {
9       console.log('%d %d', responses, res.↩
            statusCode);
10      if (++responses == N * M) {
11       console.error('Received all responses,↩
             closing server');
12       server.close();
```

Another example from this category is the following test from Node.js test[17], which, when looping rapidly and making new connections (in line 5) it can cause theRaspberry Pi 2 Model to malfunction.

```
1  const server = net.createServer(function ↩
       listener(c) {
2   connections.push(c);
3  }).listen(common.PORT, function ↩
       makeConnections() {
4   for (var i = 0; i < NUM; i++) {
5    net.connect(common.PORT, function connected↩
         () {
6     clientConnected(this);
7    });
```

Time The commits in this category have flakiness that comes from time related issues, e.g., relying on the system's time. We classify 12 commits in total into the time category. The following example[18] shows a flaky test that is the result of relying on the system's time with the use of `Date.now()` function.

```
1  const now = Date.now();
2  while (now + 10 >= Date.now());
```

The following snippet[19] from Moment can cause flakiness because of checking the equality of times that can be different by a millisecond.

```
1  test.expect(7);
2  test.equal(moment.utc().valueOf(), moment().↩
       valueOf(), "Calling moment.utc() should ↩
       default to the current time");
```

[16]https://tinyurl.com/mttkmtx5
[17]https://tinyurl.com/2rfs42zs
[18]https://tinyurl.com/354h3uf4
[19]https://tinyurl.com/mvfmf8rz

Resource Leak We classify a commit in this category in cases where the CUT did not properly manage resources, e.g., memory allocations issues, database connections, loss of connection, or not enough space. There are 10 commits from the resource leak category. The following example[20] from Node.js shows a flaky test due to `EMFILE`, which means that a process is trying to open too many files.

```
1  if (accumulated.includes('Error:') && !↩
       finished) {
2    assert(
3     accumulated.includes('ENOSPC: System limit↩
          for number ' +
4                         'of file watchers reached'↩
                              ),
5     accumulated);
```

The following snippet[21] from Node.js shows a test that is not performing proper clean-up and so it would fail if run more than one time on the same machine.

```
1  function test_up_multiple(cb) {
2   console.error('test_up_multiple');
3   if (skipSymlinks) {
4    console.log('skipping symlink test (no ↩
         privs)');
5    return runNextTest();
6   }
7   fs.mkdirSync(tmp('a'), 0755);
8   fs.mkdirSync(tmp('a/b'), 0755);
9   fs.symlinkSync('..', tmp('a/d'), 'dir');
10  @@ -432,6 +443,7 @@ function test_up_multiple(↩
       cb) {
11     if (er) throw er;
12     assert.equal(abedabed_real, real);
13     cb();
14    });
15  });
16  }
```

Other Known Causes Other remaining causes (13 commits) are related to several categories. The remaining causes are (each provided with an example from the projects we analysed): IO[22], test order dependency[23], floating point operations[24],randomness[25], and implementation dependency[26].

In addition, we categorized 7 commits in total into more than one cause. For those commits, we believe that there are multiple causes that may have an equal effect on the flaky behaviour. For example, the following test[27] from Node.js falsely assumed that closing the client (which also currently destroys the socket rather than shutting down the connection) would still leave enough time for the server side to receive the stream error.

```
1  server.on('stream', common.mustCall((stream) ↩
       => {
2   stream.on('error', common.mustCall(() => {
3   stream.on('close', common.mustCall(() => {
```

[20]https://tinyurl.com/2s44nnua
[21]https://tinyurl.com/4yycbc73
[22]https://tinyurl.com/2zv3acya
[23]https://tinyurl.com/39nnnt85
[24]https://tinyurl.com/2p999bjw
[25]https://tinyurl.com/3utzckbb
[26]https://tinyurl.com/yc2vtvzz
[27]https://tinyurl.com/2p8hwj7w

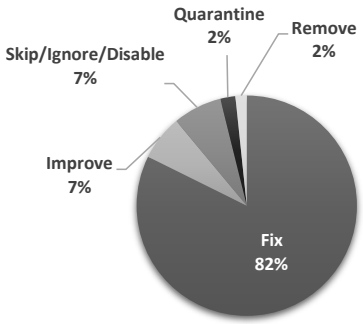

Fig. 2. Distribution of Responses Strategies

```
      server.close();
    }));
  req = client.request();
  req.resume();
  req.on('error', common.mustCall(() => {
    req.on('close', common.mustCall(() => {
      client.close();
```

Both the network and the wait for the connection issues have the same impact on the test being flaky. Hence, we categorized this test under *network* and *async wait* categories.

RQ1 findings: The top four causes of test flakiness in JavaScript projects are concurrency (21%), async wait (20%), OS (18%) and network (13%). Other causes of flaky tests we found include platform, UI, hardware, time and resource leak.

B. RQ2: What are the common strategies followed when dealing with flaky tests in JavaScript projects?

When developers face a flaky test, they usually fix it by changing the test code, the CUT, or both. We found that a total of 350 commits to fix flakiness by changing the test code, 3 commits changed CUT, and 5 commits made changes to both. Although those changes aim to fix the flaky tests, they do not necessarily completely get rid of the flaky behaviour, but instead they improve/reduce such behaviour. In the other words, the changes sometimes decrease the chance of the flaky behaviour but do not eliminate it.

For each commit, we identify the main strategies that developers use to deal with flaky tests. We categorise those into one of five different strategies, as follows. A visual summary of the distribution of commits based on the followed strategy is shown in Fig.2.

Fix: This strategy aims to fix any flaky test that has been detected/reported (after reproducing and confirming the flaky behaviour). We found that 295 commits to present an immediate fix of the noted flaky test. This is not surprising as we mainly mined commits with known flaky tests (that is, flaky tests that have already been detected by the developers, and thus it is very likely to have been actioned). The following example[28] from Node.js shows a flaky test from the *Hardware* category (the test failed due to set numbers of clients in Raspberry Pi). The CI results[29] show that it is fixed by reducing the number of clients from 100 to 16.

```
var N = 10;
var M = 10;
var N = 4;
var M = 4;
server.listen(common.PORT, function() {
  for (var i = 0; i < N; i++) {
    setTimeout(function() {
      for (var j = 0; j < M; j++) {
        http.get({ port: common.PORT, path: '/' ↩
            }, function(res) {
```

Improve: In this strategy, the test code or the CUT are changed to decrease the chance of flakiness or to make the code more traceable to fix it later (which makes it technical debt in nature). There are 23 commits that in this category. The following example[30] from Atom shows an added delay to the tests to make them less flaky, but they did not fix the root problem (an explanation is provided in the comment section in the following listing).

```
// In Windows64, in some situations nsfw (the ↩
    currently default watcher)
// does not trigger the change events if they ↩
    happen just after start watching,
// so we need to wait some time. More info: ↩
    https://github.com/atom/atom/issues/19442
await timeoutPromise(300);
```

Skip/ignore/disable: This strategy provides an option to developers to *skip* or *ignore* the test that is flaky. In some cases, especially when the developer is fully aware of the implications of those flaky tests, they may decide to *disable* those flaky tests and continue with the build as planned. We found 26 commits in total that belong to this category. For example, in the following code[31] from Electron, there is a condition to skip the test for "win32" platform or "arm64" architecture.

```
if describe(process.platform !== 'win32' || ↩
    process.arch !== 'arm64')('did-change-↩
    theme-color event', () => {
```

The developer noted the following in a pull request[32]:

"...there are a couple of tests that are consistently flaky on arm (both Linux and Windows variants). This PR disables those flakes so that we can get our ARM CI to the point where it can be relied on for PR validation instead of maintainers ignoring it."

Quarantine: In this strategy, developers isolate flaky tests from other healthy tests by keeping them in a quarantined area (e.g., different test suite or development branch) in order to diagnose and then fix those tests. This is a common strategy used in practice (e.g., [8], [34], [35]).There are 8 commits that belong to this category. For example, the code[33] from Node.js

[28] https://tinyurl.com/mttkmtx5

[29] https://tinyurl.com/2zz6fmvs
[30] https://tinyurl.com/nwmexy27
[31] https://tinyurl.com/bdzewfc7
[32] https://tinyurl.com/3epjpe5b
[33] https://tinyurl.com/4ybhbhj7

move a portion of the test to a separate file as the test is flaky on CentOS. The developer then noted that:

> *"[test] has been flaky on CentOS. This allows us to .. eliminate the cause of the flakiness without affecting other unrelated portions of the test."*

Remove: The strategy suggests that all flaky tests should be removed from the test suite. We found 6 commits that remove the test to completely eliminate the flaky behaviour. For example, there is a commit[34] from Node.js shows a tests that was completely removed the test from the test suit due to being flaky. The developer noted the following:

> *"[the test] is supposed to test an internal debug feature, but what it effectively ends up testing, is the exact lifecycle of different kinds of internal handles ... making the test fail intermittently ... It's not a good test, delete it."*

Fig.3 demonstrates the percentages of strategies followed based on the casue of flakiness. The *Fix* strategy represents the largest portion of each category. The most commits under *Improve* category are related to *UI*, *hardware*, and *time* causes. Similarly, as expected, most *Skip* actions are taken to deal with *OS* and *Platform*-dependent tests. The reason is that the tests are conditional in nature (will run in specific OSs or platforms), so that the test can run without any non-determinism. Also, the largest portion of *Remove* is related to *UI* and *Platform* categories. We list the most common change patterns used to fix the top four causes (concurrency, async wait, OS and network) in Table IV.

RQ2 findings: We found that 82% of flaky tests are fixed (to eliminate the flaky behaviour), 7% improved (reduces flakiness), 7% skipped or disabled, 2%quarantined (for a later fix), and 2% completely removed.

C. Implications

Our study has implications for both JavaScript testers (who write test code that might be flaky) and designers of flaky test detection and management tools. Our results show that JavaScript projects, as with projects in other languages studied to date, are prone to flaky tests, and that common causes of flaky tests in programs written using other languages (in particular Java and Python) are also prevalent in JavaScript. Similar to prior studies [16], [17], we observed *Concurrency* and *Async Wait* as top causes for flakiness in JavaScript projects. Unlike in other empirical studies, *Platform* is also a major cause of flakiness in JavaScript. This includes all scenarios where the test runs in a different environment, browser, or OS. The implication from this observation is that implicit platform dependence (e.g. specific OS) must be made explicit in the test code. Having automated ways of detecting platform related flaky tests would be of great value to developers. Any methods that can enhance developers' understanding of the requirements and limitations of the different platforms used (on-the-fly) can potentially reduce the number of flaky tests caused by platform dependency.

TABLE IV
COMMON CHANGES FOR FOUR TOP CATEGORIES

Category	Common Change	Description
Concurrency	waitForEvent increase timeout setTimeout setInterval sleep add lock	In some cases, the test is flaky because of conficliting operations in different threats. By making the test wait and adding locks, they can prevent race conditions and synchronizing operations.
Async Wait	waitForEvent async reordering code Await setTimeout setInterval increase timeout timeoutPromise	Most of the time, developers add time to the test to deal with async wait related test flakiness. For doing this, they usually increase time out or wait for special events.
OS	add condition async setTimeout	In most of the cases, developers check the type and version and run/skip tests on specific ones. Sometimes, the test is flaky because of the way the OS manages resources as it can take more time to run than expected.
Network	common.port socket.setTimeout socket.destory socket.end() client.shutdown()	In most cases, network related flakiness are causes by bad socket or port management, or lost of the internet connection.

The results of *Concurrency* and *Async Wait* being the most dominant cause of flakiness is expected given that JavaScript applications are highly asynchronous. Unlike previous studies [16], [24], we observed a small number of flaky tests that were caused by *Test Order Dependency*. The platform or testing frameworks are unlikely to be the cause for this low number, since testing frameworks for JavasScript (e.g. Jest[35], Mocha[36]) have support for parallelising and ordering test execution and support for shared state across tests. It is possible that implicit shared state is more likely in object-oriented languages such as Java with static fields. Note that nearly 47% of shared state in order-dependent tests [36] are external (e.g. files, database).

In addition, current detection tools focus mainly on languages like Java, Python and Ruby. Searching through the different resources and the recent literature surveys on flaky tests [36], [37], we found only one tool, NodeRacer [38], that target to manifest test flakiness causes by event races in JavaScript programs. There is related work on detecting other concurrency issues in JavaScript programs such as NRace [39] or atomicity violations, NodeAV [40]. There is, generally, a lack of research on tools or techniques to deal with non-deterministic tests in JavaScript projects. More work is needed to provide detection tools that target other categories of causes for test flakiness in JavaScript, such as order dependent, OS and network flakiness.

[34]https://tinyurl.com/3uzke4rf

[35]https://jestjs.io/

[36]https://mochajs.org/

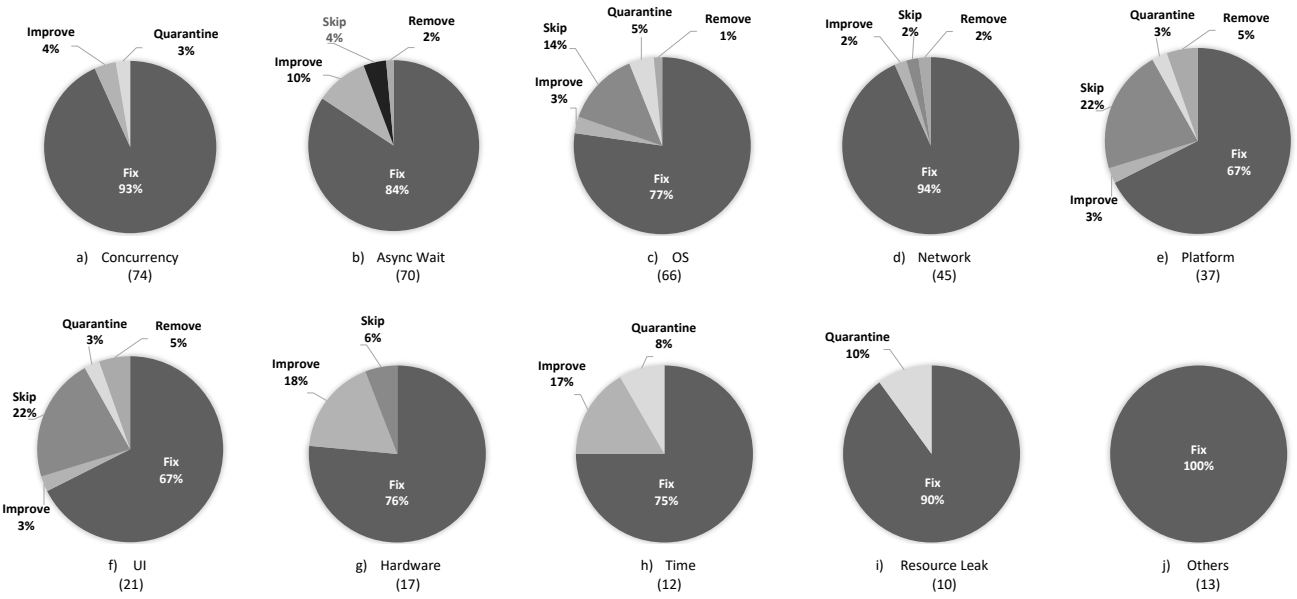

Fig. 3. Distribution of Flakiness Causes on Strategies (the number of commits are shown in parentheses)

V. Threats to Validity

Missing flaky tests related commits: We used a keyword-based approach to locate commits that are flakiness related. To provide an extended coverage, we used an extended list of 7 flaky test-related keywords in our search string (flaky, intermittent, fragile, brittle, flakey, non-deterministic test and intermit). It is still possible that we could miss flaky tests that either 1) used keywords other than the one we identified in our research, or 2) flaky tests that has no associated commits associated (only reported in the issue tracker but has not been reviewed or actioned yet).

Generalisation of the findings: The study considers commits from JavaScript projects that are publicly available on GitHub. The observations noted in this study, in terms of the categories of flaky tests, can be limited to the selected projects and may not be generalizable to other JavaScript programs. We mitigated this by selecting popular and highly starred projects that are managed by big communities and represent various types of applications (server, frontend/backend frameworks, web applications, graphics, etc.). Thus, our sample is somehow representative of real-world programs. Including additional projects will surely supplement our findings, and this may lead to more generalised conclusions.

Manual classification of causes: A large part of the study depends on manual analysis of the data (commit messages and source code changes), which could affect the construct validity due to potential personal oversight and bias. In order to reduce potential false positives, all identified commits were independently classified by two authors (all commits were separately classified by the first two authors), and when there were disagreements about a certain commit, we introduced a conflict resolution step where the third author was then involved to also independently classify the disagreed on commits. We followed that with a rigorous discussion of the causes between all authors until we reached an agreement (100% agreement level). There were still issues with a number of commits that all authors agreed that the cause of flakiness is unknown or cannot be identified - those have been classified as "hard to classify" and then excluded from the analysis.

VI. Conclusion

In this paper, we investigate the presence of flaky tests in JavaScript projects. We first categorise flaky test-related commits based on the cause of flakiness, and then investigate the common strategies followed to deal with test flakiness. Our study shows that the common causes of flaky tests in JavaScript are not different from those noted in other languages - in particular, Java [16] and Python [24]. We found that 70% of flaky test commits in JavaScript are caused by one of the following: *Concurrency*, *Async Wait*, *OS* or *Network*. Unlike in previous studies [16], [24], which identify test-order dependency as one of the key causes of flakiness, we did not find many flaky test commits that are caused by test-order-dependency. In terms of fixing strategies, we note that the majority of flaky tests (82%) are fixed to eliminate flaky behaviour completely. A smaller percentage of those flaky tests are either skipped (ignored), quarantined (to be fixed later, becoming a technical debt) or completely removed from the test suite. These results can help future research on flaky tests, in particular JavaScript tools designers in building tools that help to detect and remove flaky tests from test suites.

Acknowledgment

This work was partially funded by a SfTI National Science Challenge grant No. MAUX2004.

REFERENCES

[1] M. Fowler, "Eradicating non-determinism in tests," 2011, https://martinfowler.com/articles/nonDeterminism.html.

[2] A. Sandhu, "How to fix flaky tests," 2015. [Online]. Available: https://tech.justeattakeaway.com/2015/03/30/how-to-fix-flaky-tests/

[3] J. Palmer, "Test flakiness – methods for identifying and dealing with flaky tests," 2019. [Online]. Available: https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/

[4] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.

[5] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 298–311.

[6] B. Vancsics, T. Gergely, and Á. Beszédes, "Simulating the effect of test flakiness on fault localization effectiveness," in *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, 2020, pp. 28–35.

[7] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "On the impact of flaky tests in automated program repair," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 295–306.

[8] J. Micco. Flaky tests at google and how we mitigate them. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[9] G. Pirocanac. Test flakiness – one of the main challenges of automated testing. [Online]. Available: https://testing.googleblog.com/2020/12/test-flakiness-one-of-main-challenges.html

[10] 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)*. IEEE, 2015, pp. 101–110.

[11] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 821–830.

[12] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 1–11.

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

[14] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1479–1480.

[15] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1585–1597.

[16] 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*, 2014, pp. 643–653.

[17] 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)*. IEEE, 2018, pp. 534–538.

[18] A. Sjöbom, "Studying test flakiness in python projects: Original findings for machine learning," 2019.

[19] S. Technologies, "Why is javascript so popular?" 2018. [Online]. Available: https://web.archive.org/web/20191208013733/https://www.simplytechnologies.net/blog/2018/4/11/why-is-javascript-so-popular

[20] I. Lima, J. Silva, B. Miranda, G. Pinto, and M. d'Amorim, "Exposing bugs in javascript engines through test transplantation and differential testing," *Software Quality Journal*, vol. 29, no. 1, pp. 129–158, 2021.

[21] A. Ahmad, O. Leifler, and K. Sandahl, "Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions," *arXiv preprint arXiv:1906.00673*, 2019.

[22] 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*, 2019, pp. 830–840.

[23] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proc. ACM Program. Lang.*, vol. 4, 2020. [Online]. Available: https://doi.org/10.1145/3428270

[24] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 148–158.

[25] 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*, 2020, pp. 1471–1482.

[26] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2020, pp. 211–224.

[27] J. Morán Barbón, C. Augusto Alonso, A. Bertolino, C. A. Riva Álvarez, P. J. Tuya González *et al.*, "Flakyloc: flakiness localization for reliable test suites in web applications," *Journal of Web Engineering, 2*, 2020.

[28] 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.

[29] 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.

[30] 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*, 2019, pp. 545–555.

[31] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! detecting flaky tests caused by concurrency with shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 301–311.

[32] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in android via event order exploration," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 367–378.

[33] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in node. js," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 520–531.

[34] E. Zhu, "Solving flaky tests in rspec," 2019. [Online]. Available: https://flexport.engineering/solving-flaky-tests-in-rspec-9ceadedeaf0e

[35] R. Agarwal, "Handling Flaky Unit Tests in Java," Jun. 2021. [Online]. Available: https://eng.uber.com/handling-flaky-tests-java/

[36] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.

[37] B. Zolfaghari, R. M. Parizi, G. Srivastava, and Y. Hailemariam, "Root causing, detecting, and fixing flaky tests: State of the art and future roadmap," *Software: Practice and Experience*, vol. 51, no. 5, pp. 851–867, 2021.

[38] A. T. Endo and A. Møller, "Noderacer: Event race detection for node. js applications," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 120–130.

[39] X. Chang, W. Dou, J. Wei, T. Huang, J. Xie, Y. Deng, J. Yang, and J. Yang, "Race detection for event-driven node. js applications," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 480–491.

[40] X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, and T. Huang, "Detecting atomicity violations for event-driven node. js applications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 631–642.