The Report Committee for Gillian Ann Yost
Certifies that this is the approved version of the following Report:

Finding Flaky Tests in JavaScript Applications Using Stress and Test Suite Reordering

APPROVED BY
SUPERVISING COMMITTEE:

August Shi, Supervisor

Milos Gligoric

Finding Flaky Tests in JavaScript Applications Using Stress and Test Suite Reordering

by

Gillian Ann Yost

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2023

Dedication

To my family: my parents Dennis and Wendy Yost, brothers James and Jarad Yost, and my sister Rachel Yost for their unwavering support in completing graduate school.

Acknowledgements

This work was built on and based on a project that Rifat Seleoglu and I did in Fall 2021. With his permission I have reused/altered some of the paragraphs from our final report of that project in the background and related work sections.

Thank you to my supervisor Dr. August Shi. After taking Software Testing in a Non-Deterministic Era in Fall 2021, I wished to finish the project Rifat and I started. Dr. Shi's advice and expertise in the field of flaky tests has been invaluable while completing my report.

Thank you to Dr. Milos Gligoric for feedback on my report and help with my writing style.

Abstract

Finding Flaky Tests in JavaScript Applications Using Stress and Test Suite Reordering

Gillian Ann Yost, MSE

The University of Texas at Austin, 2023

Supervisor: August Shi

Flaky tests are an issue when software tests fail non-deterministically on the same version of code. Research has been conducted on flaky tests in Java, C/C++, and Python, but little research has been conducted in JavaScript even though it is a popular programming language. Thus, we conducted an empirical study of flaky tests found from 58 JavaScript GitHub repositories that used Jest as their testing framework. We developed a technique called StressSequence to detect flaky tests in JavaScript applications. StressSequence stresses the machine using Stress-Ng and a random sequencer for reordering test suites in Jest. StressSequence then reruns the tests multiple times under these conditions, detecting a flaky test when it observes the test both pass and fail. We evaluated StressSequence on our subjects, comparing the detected flaky tests against a baseline technique of simply rerunning tests without stress or reordering, Reruns. The subjects were run 10 times with StressSequence and 10 times each with Reruns as a comparison. We found 71 flaky tests from both StressSequence and Reruns. StressSequence found 56 and Reruns found 41 with 26 overlapping. Of the 71 flaky tests,

30 were from timeouts and 30 came from one application due to test dependencies. Of the remaining eleven flaky tests the majority of them are of the Async Wait and UI categories. Based on the manual inspection the random sequencer did not help reveal any flaky tests. Running with stress, however, did impact the flake rate of certain flaky tests and found flaky tests not found using Reruns.

Table of Contents

List of Tables

List of Figures

Introduction

Flaky tests are tests that non-deterministically pass and fail when run on the same version of code [1] [2] [3] [4]. They make it difficult for a developer to distinguish between an actual failed test due to a software bug in the source code and a test that failed because of non-deterministic execution not due to any bug in the source code. False-positive results may cause developers to lose their trust in their test suites. Flaky tests undermine regression testing efficiency [3]. Regression testing is used in continuous integration (CI) environments to make sure that a newly introduced piece of code does not introduce a bug to the source code repository. In the case where flaky tests exist in the test suite, the CI environment may halt frequently, causing software production to stop and software development companies to lose a significant amount of money.

JavaScript is a popular programming language, but there has been little research on it in terms of flaky test detection and categorizing. The only previous research found has been analyzing GitHub commit messages on flaky tests rather than running applications [2]. There is research in flaky test detection in Java and Python for order-dependent flaky tests and automatic categorizing and fixing them [4] [5]. Additionally, there has been work in utilizing stress to find more flaky tests while rerunning tests [6].

In this work, we propose a technique called StressSequence to detect flaky tests in JavaScript applications that use Jest as the testing framework [7]. StressSequence uses stress, a test suite sequencer, and reruns to detect flaky tests. StressSequence utilizes Stress-Ng [8] to stress the machine and uses Jest's custom sequencer to randomly reorder testing suites from how Jest would typically run them. StressSequence runs tests multiple times under these conditions, detecting a flaky test when observing it both pass and fail.

We ran StressSequence on 58 JavaScript applications on GitHub, rerunning the tests 10 times to detect flaky tests. We compared StressSequence against rerunning the tests normally, Reruns, as a baseline to evaluate how effective StressSequence is at detecting flaky tests. We also compared the flake rate, which is the number of times the flaky test fails out of the times we ran it. StressSequence and Reruns found 71 flaky tests across six applications.

We categorized the flaky tests based on the categories proposed by Hashemi et al. [2]. The largest categories of flaky tests found were timeout and test dependency. However, most of the timeouts occurred using StressSequence, which does stress the machine, and all test dependency tests came from one application. The next largest categories were UI and Async Wait. Concerning flaky tests that fail not due to timeout or test dependencies, running with StressSequence impacted the flake rate of four of these flaky tests. Furthermore, StressSequene found five such flaky tests that were not found using just reruns. After our manual inspection, we determined that the reordering part of StressSequence did not have an impact on detecting flaky tests.

Our data and code are publicly available [9]. The contributions from this work are as follows:

- A technique for stressing and reordering test suites for JavaScript applications that use Jest as their testing framework.
- A data set of flaky tests found in six applications found by rerunning tests in 58 applications.

Background

FLAKY TESTS

Flaky tests are software tests that fail non-deterministically and are a major issue when verifying the functionality of code [1] [2] [3] [4]. When developing software developers use regression testing which is used to certify that the application still functions after any code changes. Regression testing assumes that tests are deterministic and thus when they are not, it can have a large impact on automated testing.

The existence of flaky tests in the test suite can hinder a software development company's ability to produce high-quality software. Identifying flaky tests manually can be expensive as it takes a long time for a developer to inspect the large number of test cases in modern software repositories. Software companies are actively looking for and researching automated methods of detecting and fixing flaky tests. The first step in this automation is to study the types of flaky test found in certain domains. There is minimal research in the area of flaky tests in JavaScript.

CATEGORIES OF FLAKY TESTS

Luo et al. [1] described ten main categories of flaky tests: async wait, concurrency, test order dependency, resource leak, network, time, IO, randomness, floating point operations, unordered collections. Hashemi et al. [2] in addition to some of the previous categories also used UI, hardware, and OS when classifying flaky tests. Concurrency flaky tests are categorized as such because they are flaky due to concurrency-related issues. Async Wait is a category describing when a test makes an asynchronous call and does not wait properly. This category is a subset of concurrency but is labeled as separate due to the large number of flaky tests found in this category. OS flaky tests fail on certain OS or OS version and pass on others. UI flaky tests are flaky due to UI features. Network flaky tests fail due to connection failures or bad socket management [2]. Test order dependency is when a test passes or fails depending on the order in which tests are run [4].

JAVASCRIPT AND TYPESCRIPT

JavaScript supports object-oriented, imperative, and declarative styles and is a prototype-based, multi-paradigm, dynamic language that operates on a single thread [10]. JavaScript is a flexible language that supports different programming paradigms. The same flexibility also enables JavaScript to be used for a variety of cases. It can be used to develop a simple web script, large-scale server software, or any mobile/desktop application. Since this language has a C-style syntax and can be used to develop a wide variety of software applications, it is popular among developers, and a recent study claims it is the most popular programming language at 16 million users [11]. JavaScript also excels at event-driven programming and asynchronous programming. Many JavaScript (therefore, TypeScript) APIs utilize asynchronous features of the language in order to not block the program execution when dealing with static resources, sending requests over the network, etc. If a developer or software development organization need to decide on a programming language and their project requires handling a variety of events in an asynchronous way or making lots of asynchronous function/API calls, then JavaScript is a suitable language for their needs.

TypeScript is a strongly typed version of the JavaScript language. It supports all the features of JavaScript, and it extends JavaScript by introducing a type system. TypeScript's primary advantage is its ability to detect unexpected behavior in code, which reduces the likelihood of bugs [12]. In this work when referring to JavaScript, we are additionally referring to TypeScript.

JEST

Testing a JavaScript application involves similar steps to testing any other application developed using another language. Previously, Mocha was the most popular testing framework but now it is Jest. The main difference between Mocha and Jest is that Jest is easier to use because it comes as a ready-to-use bundle that requires little to no configuration. The situation was different and difficult for Mocha. From making assertions to mocking functions, almost every step requires the developer to find an appropriate third-party library and integrate it into the Mocha [13]. For these reasons, Jest is the most popular JavaScript testing framework right now [14].

There are a few important characteristics and features of Jest that can be used for or affects flaky test detection. Jest by default runs test files, known as test suites, in parallel, rather than sequentially. Each test suite is assigned a worker, but the tests within a suite are run sequentially. Jest guarantees that tests in a test suite are executed in the order they are defined in the file. This characteristic may reduce the number of flaky tests, particularly order-dependent flaky tests that exist in Jest test suites. There is an option that the user can limit the number of workers or use the “runInBand” option to have the test suites run sequentially. Furthermore, if Jest determines that it will be faster to run test suites sequentially rather than in parallel, it will do so without telling the developer, which adds a level on non-determinism to the order of test suites and how they are run [15].

Jest also orders test suites according to three factors: failure of the test suite, the duration it takes a test suite to run, and test suite file size if it does not have any information on the first two factors [16]. Since these factors can change at any time, it is likely that test suites are run in a different order almost in a non-deterministic sense. Jest also comes with a test sequencer feature that allows the developer to determine the order of the test suites. This option is available for Jest versions 24.7 or newer as it was introduced in April 2019 [7]. It allows the user to override the default test sequencer with their own written one in JavaScript. This test sequencer is used for ordering tests run in parallel and run sequentially. Jest also has a default test timeout of 5 seconds but can be changed using the jest config file, the package.json or using setTimtout() function for the specific test [17]. The sequencer, running in parallel, and default timer could lead itself to flaky tests.

StressSequence

We propose StressSequence, a technique to rerun the tests while stressing the machine and randomly reordering test suite run orders. After rerunning the tests, StressSequence parses the test result failures, detecting flaky tests as the ones with both passing and failing outcomes. Figure 1 shows this technique. Stressing a machine can lead to a greater detection rate of flaky tests [6] and reordering test suites can lead to finding order-dependent flaky tests [4].

STRESS-NG

Stress-Ng is a tool to generate stress to load and stress kernel interfaces. There are various classes called stressors that are different stress mechanisms. There are 13 classes of stressors [8]. The stressors selected for this project were [18]:

- cpu: CPU intensive
- io: generic input/output
- filesystem: file system activity
- network: TCP/IP, UDP and UNIX domain socket stressors

These stressors were selected because Hashemi et al.'s empirical study specified that that Concurrency, Async Wait, OS, and Network were the top four cause of flakiness found in JavaScript applications [2]. The intuition is that stressing the CPU and the network will make concurrency, async wait, and network flaky tests easier to find. I/O and filesystem are also used with the goal of finding tests that may rely on I/O or files. StressSequence uses all stressors in these classes. There was a limit on how many classes could be selected as the stressors increased testing time and some of the classes, such as scheduler, drastically increased testing time.

Customized Jest Sequencer

StressSequence uses a custom test sequencer that utilizes the Durstenfeld Shuffle Algorithm to shuffle the array of test suites [19]. Since the random function in the Math library in JavaScript cannot be user seeded, the seedrandom package from npm was used instead [20]. The test sequencer is given the test suites as an array, shuffles them, and returns the array for Jest to begin running the tests. Every time the application's tests are run the seedrandom number generator is initialized with a new random seed generated by the default random function in the Math library as every new run is a new instance of Jest.

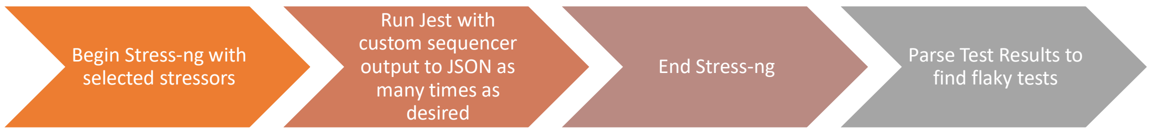
Begin Stress-ng with selected stressors
Run Jest with custom sequencer output to JSON as many times as desired
End Stress-ng
Parse Test Results to find flaky tests

Figure 1: Flaky Test Detection Technique StressSequence

Execution and Parsing

The StressSequence is configurable to rerun the application N number of times. All test reports are JSON files, which are saved together with a log of the console output. The output JSON files are parsed for the results to determine whether a test is flaky or not as the files have each test and a status of "failed", "passed" and "pending". If a test is pending the developer labeled it as such and therefore, they are pending for all runs, so only status of “failed” and “passed” could possibly change. When parsing, if a test has a different result from the very first run, it is considered flaky. If all tests fail it is not considered flaky as there is no change in result.

Methodology/Experimental Setup

Selecting Subjects

We first started by looking at the top 100 starred JavaScript applications on GitHub that depend on Jest. Then, to add some more subjects, we web scraped the applications that depended on Jest from Jest’s GitHub page. We selected the first 100 applications with 500 or more stars. Thus, a range of extremely popular more recent applications and older but popular applications are collected.

Next, we ran the tests of the selected GitHub applications, and filtered them. We only considered applications that could successfully run using default Jest and passed more than one test as subjects. 28 subjects came from the top 100 stared JavaScript applications and 30 came from the first 100 applications with 500 stars putting the total at 58. We collected the subjects on Sept 4th, 2022, and recorded the git commits, shown in the Appendix. We ran those specific git commits for the remainder of the project to ensure the same tests ran and were unchanged throughout the procedure. In the end, we selected and ran 58 subjects shown in Table 3 in the Appendix. Since the ability to reorder test suites is only in versions 24.7 and newer of Jest, 18 of the applications were older and unable to run with the sequencer because they did not support it.

Docker

Docker was used to set up the environment and have a consistent environment across different machines [21]. We used Lts-fermium, which is a version of a node image which allows for a consistent JavaScript virtual machine across devices. This image is a lightweight GNU/Linux OS with some additional dependencies installed such as node and npm. Furthermore, when running Docker, we limited the RAM to 8 GB and the CPU to 2 cores. This goal was to mimic the GitHub hosted runners which is 2-core and 7 GB of

RAM for Linux Machines [22] [23]. Hard disk memory was not inherently limited except for the capacity of the machine used.

Execution

We ran each of the 58 subjects 10 times using StressSequence, and 10 each with just rerunning the tests with no stress or sequencer, Reruns, for comparison. We selected 10 times due to time constraints on the machine used. The running methodology for StressSequence and Reruns is shown in Figure 2.

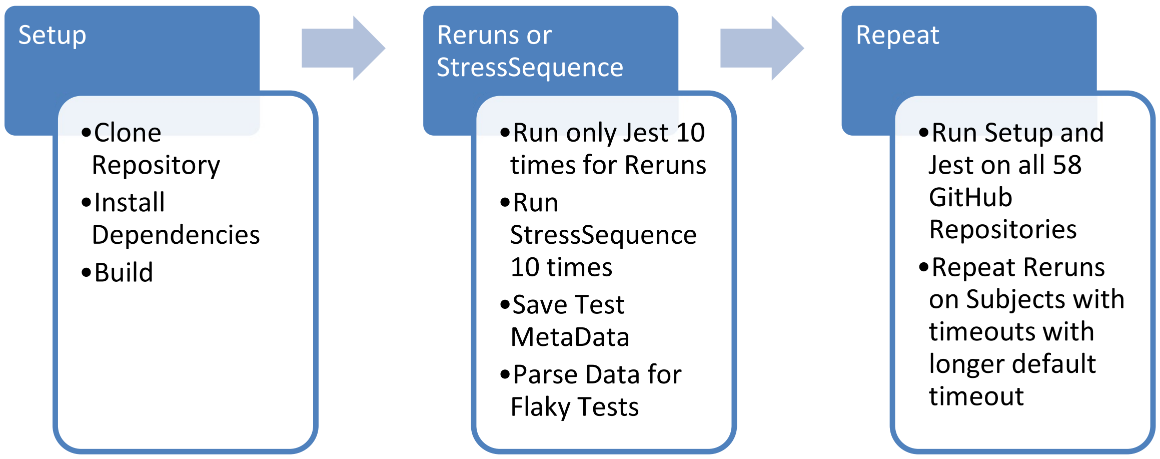
Setup
•Clone Repository
•Install Dependencies
•Build
Reruns or StressSequence
•Run only Jest 10 times for Reruns
•Run StressSequence 10 times
•Save Test MetaData
•Parse Data for Flaky Tests
Repeat
•Run Setup and Jest on all 58 GitHub Repositories
•Repeat Reruns on Subjects with timeouts with longer default timeout

Figure 2: Experimental Execution for StressSequence and Reruns

Results

RQ1: How many flaky tests were found in the JavaScript Applications?

RQ2: What were the categories of flaky tests found?

RQ3: What are the differences in flaky tests detected using StressSequence and Reruns?

RQ1 is asking how prevalent flaky tests are in JavaScript applications. It is important to know how many flaky tests there are to understand how large of an issue flaky tests are in the JavaScript domain. RQ2 is asking about the different categories of the flaky tests found. Understanding the categories of flaky tests found is the first step in automating flaky test detection and fixing the tests. Finally, RQ3 is asking is StressSequence helpful in finding flaky tests. Understanding if StressSequence helps find flaky tests tells us how it could be used in the future.

RQ1: How many Flaky tests were found?

Table 1 shows the results of running StressSequence on the 58 Subjects and the flaky tests found. Overall, we detected 71 flaky tests, and Table 1 shows the breakdown of flaky tests detected by StressSequence, Reruns, or both. Adazzle/react-data-grid and Amzn/style-dictionary had the largest number of flaky tests. These tests will be further discussed in later sections. All six applications have 2,161 tests, with 71 of them being flaky, meaning roughly 3.3% of tests were found to be flaky.

RQ1: We detected 71 flaky tests, with StressSequence detecting 56 flaky tests and Reruns detecting 41.

RQ2: What were the categories of flaky tests found?

The categories of the flaky tests found were Async Wait, UI, Network, and Test Dependency, which has previously been discussed in other research [4] [5]. However, there were tests that timeout and one flaky test where the error comes from the precision of time used for comparison. The classification of some of the flaky tests and explanations are shown in this section. For full breakdowns, see Table 2, which shows the number of flaky tests in each category for each subject.

Subject	StressSequence	Reruns	Both	**Total**
adazzle/react-data-grid	20	1	9	30
algolia/algoliasearch-client-javascript	1	2	2	5
alibaba/anyproxy	1	0	0	1
amzn/style-dictionary	6	10	14	30
atlassian/react-beautiful-dnd	1	2	1	4
babel/preset-modules	1	0	0	1
Total	30	15	26	71

Table 1: Flaky Test Results for StressSequence and Reruns

Subject	Async Wait	UI	Network	Precision Error	Timeout	Test Dependency	**Total**
adazzle/react-data-grid	0	5	0	0	25	0	30
algolia/algoliasearch-client-javascript	4	0	1	0	0	0	5
alibaba/anyproxy	0	0	0	0	1	0	1
amzn/style-dictionary	0	0	0	0	0	30	30
atlassian/react-beautiful-dnd	0	0	0	1	3	0	4
babel/preset-modules	0	0	0	0	1	0	1
Total	4	5	1	1	30	30	71

Table 2: Flaky Test Categories

Async Wait

Flaky Test Example 1

The first example of an Async Wait is from algolia/algoliasearch-client-javascript shown in the code in Figure 3 [24].

```
 it('connection timeouts with the given 1 seconds connection timeout',
     async () => {
  const before = Date.now();
  const response = await requester.send({
    ...timeoutRequest,
    ...{ connectTimeout: 1, url: 'http://www.google.com:81' },
  });

  const now = Date.now();

  expect(response.content).toBe('Connection timeout');
  expect(now - before).toBeGreaterThan(999);
  expect(now - before).toBeLessThan(1200);
});
```

Figure 3: algolia/algoliasearch-client-javascript Flaky test Async Wait Example 1 Connection Timeout Checker

The flaky test shown in Figure 3 checks that timeouts happen with the given amount of time to a url that does not exist. When returned it sometimes fails the "expect(now-before).toBeLessThan(1200);" on both StressSequence and Reruns. The margin of tolerance is 200 ms. Utilizing StressSequence, it exceeds this time 8/10 runs with return times between 1207 ms and 1900 ms. With Reruns, the test fails 6/10 times with return times between 1200 ms and 1397 ms. While taking longer than expected on the StressSequence run is unsurprising, failing more than 50% of the time using Reruns means that the tolerance may need to be increased for this test. To fix this flaky test or decrease the flake rate, the range of time should be increased 1300 or 1400 ms. Changing the value to 1300 would cut the failure rate of the 10 runs in half, putting it at 3/10, and 1400 would have no failures. There are two other flaky tests similar to this nature found in this application.

Flaky Test Example 2:

```
it('respects TTL', async () => {
    // Set one host down.
    await transporter.hostsCache.set(transporter.hosts[0], {
      ...createStatefulHost(transporter.hosts[0], HostStatusEnum.Down),
      lastUpdate: Date.now() - 60 * 2 * 1000 + 10, // should be down
    });

    expect(
      (await createRetryableOptions(transporter.hostsCache, trans
      porter.hosts)).statelessHosts
    ).toHaveLength(2);

  ...

});
```

Figure 4: algolia/algoliasearch-client-javascript Flaky test Async Wait Example 2 Resepcts TTL

This flaky test shown in Figure 4 from the same subject assesses that the transporter respects the time to live in a router [24]. The transporter host array holds three stateless hosts, [“read.com”, “write.com”, “read-write.com”]. A requester is made before each test that the transporter uses. This is shown in Figure 5. A stateful host consists of a stateless host, a lastUpdate value, and a status (up or down). These are held in the hostsCache. This test fails once in 10 runs, because in the first check of length in the createRetryableOptions.statelessHosts is supposed to read 2, but in the failure message the length is 3 and consists of all three hosts. This flaky test was only found in the StressSequence run.

This test starts with all three hosts being stateless. The test then makes the first host (host 0) stateful by changing the last updated value and status as down in the cache. When creating the retriable shown in Figure 6 and determining stateless hosts, it checks if each host is down and timed out. The lastupdate value of host 0 is set to 10ms away from not being considered a timeout. If the test takes longer than 10ms to get to the isStatefulHostTimeouted in the creation of the retriable options on line 3 in Figure 4 then host[0], which was made stateful, is not considered stateful anymore. Thus, now hosts are added to hostsAvailable, which means that the condition hostsAvailbale.length > 0 is false and all hosts are returned. When all hosts are returned the length checker fails. If it takes less than 10 ms then host[0] is considered timed out and thus is added to hostsAvailable meaning that the two other hosts are returned as stateless hosts.

```
beforeEach(() => {
    const requester = createFakeRequester();
    requesterMock = spy(requester);
    transporter = createFixtures().transporter(requester);

    when(requesterMock.send(anything())).thenResolve({
      content: '{"hits": [{"name": "Star Wars"}]}',
      status: 200,
      isTimedOut: false,
    });
  });
```

Figure 5: algolia/algoliasearch-client-javascript Flaky test Async Wait Example 2 Resepcts TTL Before Each Test

```
 const hostsUp = statefulHosts.filter(host => isStatefulHostUp(host));
    const hostsTimeouted = statefulHosts.filter(host => isState
       fulHostTimeouted(host));

    /**
     * Note, we put the hosts that previously timeouted on the end of the list.
     */
    const hostsAvailable = [...hostsUp, ...hostsTimeouted];

    const statelessHostsAvailable =
      hostsAvailable.length > 0
        ? hostsAvailable.map(host => createStatelessHost(host))
        : statelessHosts;
```

Figure 6: algolia/algoliasearch-client-javascript Flaky test Async Wait Example 2 Resepcts TTL Retryable Options Code Snippet

UI

UI flaky tests have to do with testing UI portions of JavaScript. Figure 7 shows an example flaky test from this category.

Flaky Test Example 3:

```
test('cellNavigationMode="NONE"', async () => {
    setup({ columns, rows, bottomSummaryRows, cellNavigationMode: 'NONE' });

    // pressing arrowleft on the leftmost cell does nothing
    await userEvent.tab();
    await userEvent.keyboard('{arrowdown}');
    validateCellPosition(0, 1);
    await userEvent.keyboard('{arrowleft}');
    validateCellPosition(0, 1);

    // pressing arrowright on the rightmost cell does nothing
    await userEvent.keyboard('{end}');
    validateCellPosition(6, 1);
    await userEvent.keyboard('{arrowright}');
    validateCellPosition(6, 1);

    // pressing tab on the rightmost cell navigates to the leftmost cell on the
    // next row
    await userEvent.tab();
    validateCellPosition(0, 2);

    // pressing shift+tab on the leftmost cell navigates to the rightmost cell
    // on the previous row
    await userEvent.tab({ shift: true });
    validateCellPosition(6, 1);
    });
```

Figure 7: UI Flaky Test Example from adazzle/react-data-grid Flaky Test Example 3

Figure 7 shows a flaky test from adazzle/react-data-grid that is testing the keyboard navigation of a react component that shows an integrative table or grid [25]. Certain cells are highlighted and validateCellPosition() checks to see if the row and column specified is highlighted. The table is set up at the beginning of the test and then two user events are executed. The cell position is then validated, which is where the test occasionally fails.

An asynchronous function cannot be called and return its output directly without ensuring that there is a response in the first place. Such a perfunctory implementation can introduce flaws into the source code and these flaws can create flakiness in the test results. The correct ways to receive a response from an asynchronous function are as follows: implementing a callback function and passing it to the asynchronous function, utilizing "Promise" structure, or annotating methods with the "async" keyword and prepending asynchronous statements with the "await" keyword [26] [27].

The test is a known UI flaky test error in the subject's version of the user-event package from the testing-library from React [28] [29] . JavaScript relies on promises and "await" waits until a promise is returned. The await userEvent.tab(); waits until the promise is returned, and the promise is returned when the event tab has executed. This is similar for the await userEvent.keyboard('{arrowdown}'); where the promise is returned when the arrowdown event has finished. The issue is that while the user event has occurred, the table or grid may not finish rendering before the validateCellPosition() is executed, causing the test to be flaky and only sometimes failing. The fix and standard practice for using the user-event package is to use waitFor() or wrap the section in act() [28] [29] [30].

There are 5 UI flaky tests found in adazzle/react-data-grid that are like the example in Figure 7 where a userEvent is used. These four other tests were found using StressSequence and did not appear using Reruns.

Network

One Network flaky test was found in algolia/algoliasearch-client-javascript for "error handling resolves dns not found" where the network timed out or lost connection, causing the test to fail [24]. This was found using Reruns, so the network stressors did not cause the connection to fail, it was an error with the connection itself.

Precision Error

Flaky Test Example 4:

```
it('should call the onBeforeDragState and onDragStart in the correct order',
      () => {
    let mockCalled: ?number = null;
    let onBeforeDragStartCalled: ?number = null;
    let onDragStartCalled: ?number = null;
    const mock = jest.fn().mockImplementation(() => {
        mockCalled = performance.now();
    });
    const responders: Responders = getRespondersStub();
    // $FlowFixMe - no property mockImplementation
    responders.onBeforeDragStart.mockImplementation(() => {
        onBeforeDragStartCalled = performance.now();
    });
    // $FlowFixMe - no property mockImplementation
    responders.onDragStart.mockImplementation(() => {
        onDragStartCalled = performance.now();
    });
    const store: Store = createStore(
        middleware(() => responders, getAnnounce()),
        passThrough(mock),
    );

    // first initial publish
    store.dispatch(initialPublish(initialPublishArgs));
    expect(responders.onBeforeDragStart).toHaveBeenCalledWith(getDragStart());
    // flushing onDragStart
    jest.runOnlyPendingTimers();

    // checking the order
    invariant(onBeforeDragStartCalled);
    invariant(mockCalled);
    invariant(onDragStartCalled);
    expect(mock).toHaveBeenCalledTimes(1);
    expect(onBeforeDragStartCalled).toBeLessThan(mockCalled);
    expect(mockCalled).toBeLessThan(onDragStartCalled);
    });
```

Figure 8: Precision Error Flaky Test from atlassian/react-beautiful-dnd Flaky Test Example 4

In the flaky test in Figure 8 from atlassian/react-beautiful-dnd two functions onBeforeDragStart() and onDragStart() are called in a certain order and the test checks onBeforeDragStart() is run before onDragStart() by recording the start time in milliseconds [31]. Then the start time of on onBeforeDragStartCalled is expected to be less than mockCalled. However, there are instances where the values are equal since the time measurement is in milliseconds. An example error message is "Error: expect(received).toBeLessThan(expected)\n\nExpected: < 8101 Received: 8101". Occasionally, the start times of the two are equal happening in the same millisecond, which in the realm of computing is long enough for two lines of code to execute at the same time. This test was found by both StressSequence and Reruns.

Timeout

There are 30 timeout flaky tests found where the test ran longer than the specified or default test length time. Jest has a default test timeout of 5 seconds, but specific test timeout limits can be set per test or the jest.config file can be modified to change the test timeout limit. Of these 30, 12 were found using Reruns and StressSequence, and 3 of those were only flagged as flaky from Reruns. However, those three tests time out all 10 runs under StressSequence and thus, while not labeled as flaky in that particular run, they are still considered flaky.

Utilizing Reruns, two subjects: atlassian/react-beautiful-dnd and the adazzle/react-data-grid had timeout flaky tests. When the two subjects were rerun with a new default timeout limit of 30s, all the default timeout tests were not considered flaky, and only 2 tests still were flaky with test specific timeouts. This could be because the application assumes more cores or greater RAM.

There were 19 timeout flaky tests found during only the StressSequence run. While these tests are still flaky the stress contributed to longer run times, which may have caused the timeouts. The timeout default could be increased if it is expected the tests to possibly run with stress on the machine. Also, not every test failed a timeout when run with the StressSequence technique, meaning those that did may be close to the timeout time as well.

Test Dependency

In the amzn/style-dictionary subject, 30 test dependency flaky tests were found [32]. The amzn/style-dictionary subject is a build system for creating cross-platform styles. Tests create and build files but each test uses the same "test.txt" or "test.json" with the same directory path in __tests__/__output. The tests check and delete this same file and contents of the directory. Shown in Figure 9 is the clearOutput function that is called in the beginning and end of certain tests that use this directory. It clears the __tests__/__output directory, which would delete any files other tests are using as well. The subject was originally run with the default jest command and thus the test suites were run in parallel. The test command specified in the package.json has the "runInBand" option added. When run with the methodology found previously stated with the "runInBand" is added to the default jest command as specified in their package.json, we detect no flaky tests, because the test dependencies only matter when the test suites are run in parallel. By creating unique test files and directories the dependencies between test suites are fixed and the tests can be run in parallel instead of sequentially which saves time. We created this patch and the new fork of the code can be found publicly [33]. We have submitted a pull request and are waiting for a response [34].

```
clearOutput: function() {
  fs.emptyDirSync('__tests__/__output');
},
```

Figure 9: Directory clearing function used in amzn/style-dictionary

RQ2: Flaky tests found were categorized as Async Wait, UI, Network, Precision Error, Timeout, and Test Dependency. Timeout and Test Dependency were the largest however most timeouts came from StressSequence runs. UI and Async Wait were then next two largest categories with Network and Precision Error only each having one flaky test.

RQ3: What are the differences in flaky tests detected using StressSequence and Reruns?

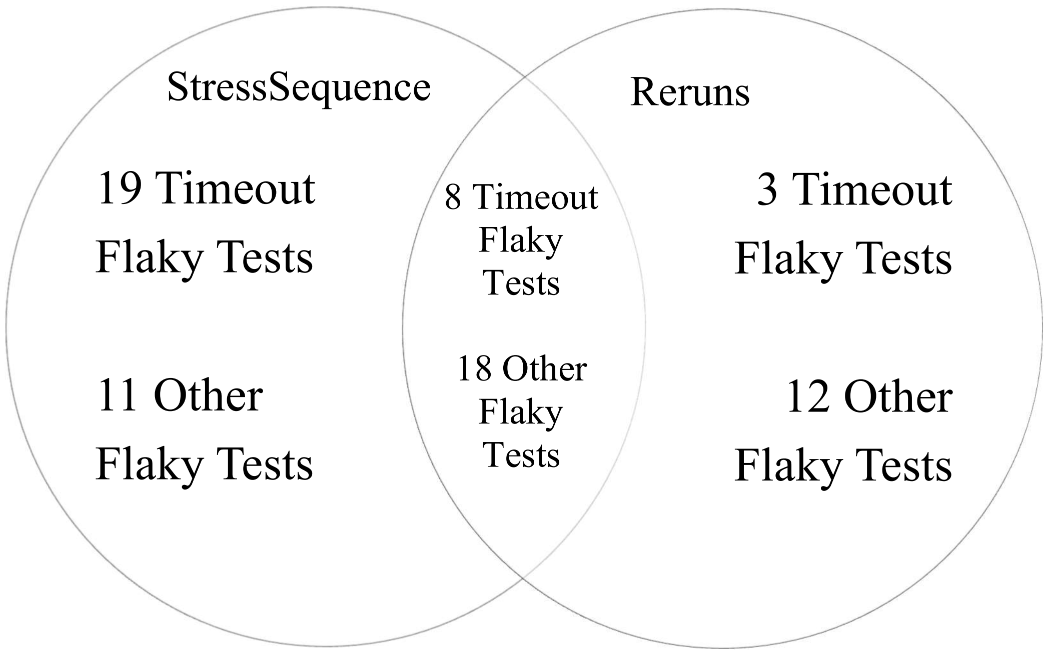

Figure 10: StressSequence versus Reruns Flaky Tests Found

The StressSequencer found more flaky tests Reruns, shown in Figure 10. The StressSequence runs found 15 more flaky tests than Reruns. However, when not considering the timeout or the test dependency subject example, StressSequnce found eight flaky tests and Reruns found six tests with three overlapping.

Tests Found by Just StressSequence

For the UI flaky tests, four of the five flaky tests were only found using StressSeqeunce. These tests are flaky because the rendering does not finish before the assertion (checker) checks the selected cell, so stress can increase the time it takes to render, increasing the likelihood of this style of flaky test failing.

Flaky Test Example 2 in Figure 4 from algolia/algoliasearch-client-javascript is only found during the StressSequence runs. It has a failure rate of 1/10 and appears when the section of code takes longer than 10 ms to execute. Adding stress can increase execution time, which could cause the flake rate to be higher.

Tests Found by Just Reruns

The Network flaky test was found only by Reruns, but its flake rate is 1/10, so it appears the network was faulty just during the time the test was run. Furthermore, there are other Async Wait flaky tests such as "browser-xhr-requester.test.ts conection timeouts with the given 2 seconds connection timeout" from algolia/algoliasearch-client-javascript that was only found using Reruns. This test is similar to one found by StressSequence, Flaky Test Example 1, and its flake rate is 1/10, meaning there is no sufficient evidence that Reruns was better for detecting this test.

Tests Found by Both StressSequence and Reruns

For Async Wait the "unit/node-http-requester.test.ts timeouts with the given 1 seconds connection" flaky test from algolia_algoliasearch-client-javascript has a failure rate of 9/10 using StressSequence versus 6/10 using Reruns [24]. This test measures the timeout time between 999 to 1200 and sometimes goes over, and it goes over more often with stress making the flaky test easier to detect.

For the single test that was found by both the StressSequence and Reruns in the UI category the failure rate was 5/10 for StressSequence and 6/10 for Reruns. However, more flaky tests of this category were found using the StressSequence method.

For the precision error flaky test, Reruns had a higher failure rate of 4/10 compared to 1/10 for StressSequence. This test fails when two functions execute in the same measured millisecond. This happens more with Reruns because the StressSequence can increase execution times, which decreases the likelihood of the two functions executing in the same measured millisecond.

RQ3: The StressSequence technique found more UI, Timeout, Async Wait, and timeout flaky tests and it also affected the flake rate of tests found by both.

Discussion

Flaky tests are prevalent in JavaScript applications, and this work found 71 flaky tests looking at 58 applications. The largest category of flaky tests found were timeouts. However, that is because most of the timeouts occurred during StressSequence runs, which stresses the machine and increases run time. The next largest category was Test Dependency, but those tests were found in only one subject and were found because the programmer's test command was not run. The tests were improved by removing the dependencies and now the test suites can be run in parallel, the default configuration of Jest. Of the final eleven flaky tests, ten out of eleven were related to timing or execution time. Based on the tests found and analyzed, the sequencer did not help in finding flaky tests. Jest only reorders test suites, so ideally when a programmer is writing tests, suites typically do not have any dependencies between them.

Concerning detecting flaky tests, starting by running the tests with only rerunning will find some flaky tests, but adding stress will help find flaky tests related to execution time. Four out of five UI tests were found using stress, so it can be helpful in finding certain types of flaky tests.

Overall, the test suite reordering did not have any impact on or improve flaky-test detection. Jest can only re-sequence test suites and guarantees that tests in a test suite are run in the order they are written. Tests suites typically not having dependencies between them is a reasonable outcome. Furthermore, the only order-dependent tests found in amzn/style-dictionary were the 24 tests detected with Reruns versus the 20 tests detected with StressSequence.

Threats to Validity

Our results may not generalize to all JavaScript applications in the world. Instead, a diverse group of 58 popular GitHub repositories that each exceeded 500 stars were used to represent JavaScript applications. Thus, we believe our results to be representative of popular JavaScript applications.

Due to limited resources, we only ran the tests within the 58 projects 10 times for both StressSequence and Reruns. From previous research, it can be difficult to detect non order-dependent tests as the probability of them occurring is small and some works could not even find all the flaky tests when even running 10,000 times [35]. Therefore, we cannot say that the flaky tests found are all the flaky tests in the applications and there may be some not found.

Finally, we ran each subject using the default jest command instead of the programmer's test command in the package.json file. This could possibly impact the number of flaky tests found in the subjects. It did impact amzn/style-dictionary which specified the "runInBand" command and using the given test command would not have flagged any flaky tests.

Related Work

Previous works have researched flaky-test detection techniques that helped define the scope and direction of this research. Several researchers have done research relating to flaky tests, detecting flaky tests, and order-dependent tests.

In one of the first papers published on flaky tests, Luo et al. [1] found that Async Wait, Concurrency, and test order dependency were the top three causes of flaky tests. They found 74 Async Wait flaky tests commits, which is a subset of Concurrency but labeled separately due to the large amount found that are related to asynchronous calls. The Async Wait categories were fixed by using/modifying waitFor or adding/modifying sleep. 32 commits were categorized as Concurrency because their flakiness is related to different threads interacting. The Concurrency flaky tests were mostly fixed by using atomic lock operations or making the code deterministic. They found three different types of order-dependent flaky tests. The first is several tests access the same static field without restoring the field, the second is where a shared static field is declared in the CUT rather than test code itself, and the third is external dependency. External dependencies are caused by something such as a shared file or network port, and Luo et al. found that more than half were caused by external dependencies, meaning that to find these researchers would need to rerun tests with different orders. The order-dependent flaky tests were fixed mostly by setting up or cleaning up the states that the tests shared with some fixed by removing the dependency or merging the tests. This work while analyzing and categorizing flaky tests, do not run the applications, and only looked at GitHub commits. Thus, only tests that have been noticed by programmers were included.

Lam et al. [4] created a tool for finding and classifying order-dependent flaky tests in Java called iDFlakies. The tool runs multiple configurations that reorder tests randomly, in reversed class-method, reversed-class, and the original order depending on the user input. They found that randomly reordering tests resulted in the greatest number of order-dependent flaky tests detected. In this work the test suites are randomly reordered based off iDFlakies.

Wang et al. expanded on the iDFlakies to iPFlakies using pytest to detect and automatically classify and fix order-dependent flaky tests found in Python applications [5]. They found research in other languages of flaky test lacking and thus decided to work on Python to fill the gaps. This work focuses on other languages that have a lack of research and expanding on work from other domains.

Hashemi et al. conducted an empirical study on JavaScript applications by looking at past GitHub commits from 40 popular JavaScript applications [2]. 70% of the flaky test were caused by Concurrency, Async Wait, OS and Network with 5.9% found to be caused by UI. They found little test order dependency flaky tests. They conducted an empirical study on flaky tests in JavaScript applications. However, our work ran tests in open-source repositories to find flaky tests rather than looking through GitHub commits like Hashemi et al. did for their work.

Silva et al. proposed a tool called Shaker for detecting flaky test by rerunning and adding noise in the execution environment [6]. The authors evaluated Shaker on multiple real-world applications and found it effective in detecting flaky tests resulting from concurrency. This was the research that inspired the idea to use Stress-Ng for this work as they used Stress-Ng to create Shaker.

Terragni et al. [36] proposed a new methodology for finding the root cause of flakiness in flaky tests. The proposed methodology includes a container-based architecture that allows the execution of tests in a reproducible and isolated environment, as well as the storage of execution traces for further analysis. The fuzzy-based approach is used to cluster the execution traces and identify the most common patterns that could explain flakiness. An example cluster talked about would be multi-threaded execution cluster, which containers would vary the execution architecture. In our work, we used Docker, which creates a consistent testing environment that can be modified and could be used in the future to run different testing environments.

Future Work

In the future, this empirical study could be improved by running the programmer's test command rather than the default jest command to see if that yields more flaky tests. Furthermore, running more than 10 times in a row for StressSequence and Reruns could also yield to finding more flaky tests.

Another large facet is finding a way to automate classifying the flaky test rather than relying on manually inspecting the tests, which can take some time. We could work in the future with leveraging Stress-Ng and the different types of stressors to try and auto classify or help classify the test.

Furthermore, based on Hashemi et al.'s work, different docker images with different OS's or using different platforms could also be used for finding more flaky tests, since those were larger categories of flaky tests found in their empirical study on JavaScript applications [2]. Additionally, based on Terragni et al.'s work and idea of different execution clusters testing different testing environments and configurations to better induce flaky tests, we could expand the use of Docker to try different environments as to detect more flaky tests [36].

Conclusion

We propose StressSequence, a technique to detect flaky tests in JavaScript applications that use the Jest testing framework by stressing the machine and reordering the test suite. We discovered 71 flaky tests from 58 GitHub repositories. We ran these GitHub repositories 10 times with StressSequence and 10 times each with Reruns as a control. Next, flaky subjects with timeouts found by Reruns were run again 10 times with an increased default timeout to determine if the timeout flaky tests were flaky due to other reasons besides timeout. There were no additional flaky tests found in the extended timeout length runs. From the 71 flaky tests, 30 were from timeouts and 30 came from one application due to test dependencies. Of the remaining eleven flaky tests the majority came from Async Wait and UI. Based on our manual inspection, running with the test suite reordering did not reveal any flaky tests. Running with stress, however, did impact the flake rate of certain flaky tests and found flaky tests not found using Reruns. Our code and data are publicly available [9].

From the flaky tests found it is shown that the default timeout value in Jest has caused tests to timeout, even with Reruns. This would mean those applications should either increase the default timeout value or they expect a faster hardware configuration than the one used in this work. The test dependency flaky tests were a result of Jest running tests in parallel as default. Of the leftover eleven, running StressSequence found eight of them with regular running only finding six. StressSequence helped find more flaky tests especially in the Async Wait and UI categories, so utilizing StressSequence can help detect more flaky tests than typical reruns.

Appendix

Subject	Commit Number	Description
UXPin/adele	8f93337	Design systems andpattern libraries
Airtable/airtable	97cc08e	Official Airtable JavaScript library
algolia/algoliasearch-client-javascript	b906cbf0	Algolia Search API
alibaba/anyproxy	b93f948	Configurable HTTP/HTTPS proxy
apify/apify-js	c70fdd66	Web scraping and browser automation library
apollographql/apollo-cache-persist	d536c74	Apollo Client 3.0 cache implementations
apollographql/apollo-client-devtools	3f6e0e6	Apollo Client Browser Devtools
antvis/AVA	9d224fb2	A framework for automated visual analytics
aws/aws-cdk	86fcd4f20	AWS Cloud Development Kit (AWS CDK) is an open-source software development framework to define cloud infrastructure
pixielabs/cavy	bd82093	Cross-platform, integration test framework for React Native

aws-actions/configure-aws-credentials	67fbcbb	Configure AWS credential and region environment variables for use in other GitHub Actions
CVarisco/create-component-app*	67ae259	A tool to generate different types of React components from the terminal
ds300/derivablejs*	f334daa	Library for deliverable which is an Observable-like state container with superpowers
DextApp/dext*	50713d8	Dext is a JavaScript powered smart launcher
emotion-js/facepaint*	a0166db	CCS in JS generator
artsy/fresnel	4355a07	Library for React to write responsive components to use media to adjust the display when certain conditions are met
threepointone/glam*	9fdabd4	ccs in JS for react
2fd/graphdoc	8be9dbf	Static page generator for documenting GraphQL Schema
actions/javascript-action	c06df86	Template to bootstrap the creation of a JavaScript action
Automattic/jetpack	ed06ef6596	Security, performance, marketing, and design tools — Jetpack is made by WordPress

		experts to make WP sites safer and faster, and help you grow your traffic.
auth0/lock*	6ce1355b	framework for login for auth0
arnog/mathlive	c2d47d5d	Library for A Web Component for Math Input
cyrilwanner/next-compose-plugins	d81db51	API for enabling and configuring plugins
pivotal-cf/pivotal-ui	65f68b4d	Pivotal UI is Pivotal's design system & component library. It contains CSS & React components that are styled for the Pivotal brand
babel/preset-modules	488d219	A Babel preset that enables async/await, Tagged Templates, arrow functions, destructured and rest parameters, and more in all modern browsers
kozhevnikov/proxymise	d23e0a3	Chainable Promise Proxy
alibaba/rax	72f5b35f	Rax is a progressive framework for building universal applications

atlassian/react-beautiful-dnd	1a380855	Library for Beautiful and accessible drag and drop for lists with React
amaroteam/react-credit-cards	218a7eb	Library for A slick credit card component for React.
adazzle/react-data-grid	f95c7e25	Feature-rich and customizable data grid React component/ Library
arqex/react-datetime	8071a79	Library for a date and time picker in the same React.js component
anthonyjgrove/react-google-login	7db5b96	A Google oAUth Sign-in / Log-in Component for React
asseinfo/react-kanban	9e0c9c8	Kanban/Trello board library for React.
fakiolinho/react-loading*	4982465	React component for loading animations
leebyron/react-loops	18ee1a1	React Loops work alongside React Hooks as part of the novel React Velcro architecture for building sticky, secure user interfaces that don't come apart under pressure
rcaferati/react-native-really-awesome-button	b8c856d	React Native button component

ds300/react-native-typescript-transformer*	369a457	Seamlessly use TypeScript with react-native
ReactPrimer/ReactPrimer*	2a0b835	React Primer is a component prototyping tool that generates fully connected class component code.
appleboy/react-recaptcha*	50d7246	Library for a react.js reCAPTCHA V2 for Google
airbnb/react-sketchapp*	b238e69	render React components to Sketch
akiran/react-slick	b9302d6	React carousel component
felixrieseberg/React-Spreadsheet-Component*	1980293	Spreadsheet Component for ReactJS
ReactTraining/react-stdio*	6948925	Render React.js components on any backend
Andarist/react-textarea-autosize	cd87f81	Component for React which grows with content
AlecAivazis/redux-responsive*	a546003	A redux reducer for managing the responsive state of your application for React
clarus/redux-ship*	2364ace	Side effects with snapshots for Redux
realadvisor/rifm	b51d843	React Input Format & Mask, tiny (?800b) component to transform

		any input component into formatted or masked input. Supports number, date, phone, currency, credit card, etc
zeke/semantic-pull-requests	f4916f4	GitHub status check that ensures your pull requests follow the Conventional Commits spec
sheinsight/shineout	04bf2e3f	A components library for React
amzn/style-dictionary	28787be	A build system for creating cross-platform styles
sindu12jun/table-dragger	1dcbed9	UI Table Library to drag and drop tables
alexreardon/tiny-invariant	31cf8fb	A Library for a small invarient which is A way to provide descriptive errors in development but generic errors in production
aweary/tinytime*	656cfc0	Library for a straightforward date and time formatter in <800b
atomiks/tippyjs-react	2699f04	React component for Tippy.js, Tippy is a complete tooltip, popover, dropdown, and menu solution for the web

angular-ui/ui-router	505e7fb	Angular UI-Router is a client-side Single Page Application routing framework for AngularJS
akxcv/vuera*	3ea31d3	Use Vue components in your React app and use React components in your Vue app
Armour/vue-typescript-admin-template	bff1343	A production-ready front-end solution for admin interfaces based on vue, typescript and UI Toolkit element-ui
lukeraymonddowning/whenipress	294b7f1	Library for a tiny, powerful and declarative wrapper around keyboard bindings in JavaScript

Table 3: List of JavaScript and TypeScript Test Subjects *Unable to Run Custom Sequencer. Link to application found by using github.com/[Application Name] Descriptions come from the application's GitHub.

References

[1] Q. Luo, F. Hariri, L. Eloussi and D. Marinov, "An empirical analysis of flaky tests," *Proceedings of the 22nd ACM International Symposium on Foundations of Software Engineering (SIGSOFT),* 2014.

[2] N. Hashemi, A. Tahir and S. Rasheed, "An Empirical Study of Flaky Tests in JavaScript," *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME),* Oct 2022.

[3] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov and J. Bell, "A large-scale longitudinal study of flaky tests," *Proceedings of the ACM on Programming Languages (PACMPL),* vol. 4, no. OOPSLA, pp. 1-29, 2020.

[4] W. Lam, R. Oei, A. Shi, D. Marinov and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," *12th IEEE Conference on Software Testing, Validation and Verification (ICST),* 2019.

[5] R. Wang, Y. Chen and W. Lam, "iPFlakies: A Framework for Detecting and Fixing Python," *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE Companion),* May 2022.

[6] D. Silva, L. Teixeira and M. d'Amorim, "Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker," *Proceedings of the 22nd ACM International Symposium on Foundations of Software Engineering (ICSME),* 2020.

[7] Facebook (Meta), "Facebook/jest: Delightful JavaScript testing," [Online]. Available: https://github.com/facebook/jest. [Accessed September 2021].

[8] Open Source, "Ubuntu Wiki," October 2020. [Online]. Available: https://wiki.ubuntu.com/Kernel/Reference/stress-ng#:~:text=Stress%2Dng%20measures%20a%20stress,as%20an%20accurate%20benchmarking%20figure.. [Accessed September 2022].

[9] G. Yost, "gillianyost/JavaScriptFlakiesDetector," GitHub, 2022 November 2022. [Online]. Available: https://github.com/gillianyost/JavaScriptFlakiesDetector.

[10] Mozilla, "JavaScript," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Accessed 28 March 2023].

[11] O. Hughes, "Top programming languages: Most popular and fastest growing choice for developers," November 2021. [Online]. Available: https://www.zdnet.com/article/top-programming-languages-most-popular-and-fastest-growing-choices-for-developers/. [Accessed 28 March 2023].

[12] Open Source, "Documentation - typescript for JavaScript programmers.," December 2021. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. [Accessed 28 March 2023].

[13] OpenJS, "Mocha - the Fun, Simple, Flexible JavaScript Test Framework.," Mochaja.org, 2023. [Online]. Available: https://mochajs.org/#getting-started. [Accessed 15 April 2023].

[14] M. Taleb, "JavaScript unit testing frameworks in 2022: A comparison," Raygun Blog, 6 October 2022. [Online]. Available: https://raygun.com/blog/javascript-unit-testing-frameworks/. [Accessed 15 April 2023].

[15] Facebook Open Source, "Getting started · jest.," Meta, October 2021. [Online]. Available: https://jestjs.io/docs/getting-started. [Accessed 11 December 2021].

[16] Facebook Open Source, "Configuring Jest," Meta, 2023. [Online]. Available: https://jestjs.io/docs/configuration. [Accessed 15 April 2023].

[17] Facebook Open Source, "Globals," Meta, 2023. [Online]. Available: https://jestjs.io/docs/api. [Accessed 15 April 2023].

[18] C. I. King, "Stress-Ng (Stress next Generation)," GitHub, 2023. [Online]. Available: https://github.com/ColinIanKing/stress-ng. [Accessed 14 April 2023].

[19] hisam, "Durstenfeld Shuffle Algorithm : Optimized Fisher-Yates," June 2021. [Online]. Available: https://www.thiscodeworks.com/durstenfeld-shuffle-algorithm-optimized-fisher-yates-javascript-vanilla-sort-randomize-array/60cd9c0b0c21d80014540536. [Accessed October 2021].

[20] D. Bau, "Seedrandom," September 2019. [Online]. Available: https://www.npmjs.com/package/seedrandom. [Accessed October 2021].

[21] Docker Inc, "Docker," [Online]. Available: https://www.docker.com/ . [Accessed September 2021].

[22] GitHub Inc., "About GitHub-hosted runners," 2023. [Online]. Available: About GitHub-hosted runners. [Accessed 28 March 2023].

[23] Travis CI, "Travis CI Documentation," Docs.travis-Ci, [Online]. Available: https://docs.travis-ci.com/user/reference/overview/#what-infrastructure-is-my-environment-running-on. [Accessed 14 April 2023].

[24] Algolia, "GitHub - algolia/algoliasearch-client-javascript at b906cbf0649a7a556b055a9bd3565886cd71d372," [Online]. Available: https://github.com/algolia/algoliasearch-client-javascript/tree/b906cbf0649a7a556b055a9bd3565886cd71d372. [Accessed March 2023].

[25] Adazzle, "GitHub - adazzle/react-data-grid at f95c7e25639f52f6d8dfcfebde9db30d5bd77111," GitHub, [Online]. Available: https://github.com/adazzle/react-data-grid/tree/f95c7e25639f52f6d8dfcfebde9db30d5bd77111. [Accessed March 2023].

[26] Mozilla, "Introducing asynchronous JavaScript - learn web development: MDN.," [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing. [Accessed 28 March 2023].

[27] Mozilla, "await," mdm web docs, 4 April 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await. [Accessed 12 April 2023].

[28] jessethomson, calebeby, gcvfi and kentcdodds, "userEvent.click does not wait for useEffect changes to be flushed #255," GitHub, 1 May 2020. [Online]. Available: https://github.com/testing-library/user-event/issues/255. [Accessed 26 March 2023].

[29] tgandrews, Gpx, pietmichal, calebeby, kentcdodds, afontcu and babramczyk, "Error thrown when using react hooks #128," GitHub, 6 June 2019. [Online]. Available: https://github.com/testing-library/user-event/issues/128. [Accessed 13 April 2023].

[30] D. Cai, "React Testing Library and the "not wrapped in act" Errors," Medium, 29 April 2020. [Online]. Available: https://davidwcai.medium.com/react-testing-library-and-the-not-wrapped-in-act-errors-491a5629193b. [Accessed 15 April 2023].

[31] Atlassian, "GitHub - atlassian/react-beautiful-dnd at 1a380855d0d008c5d6ef8e34c7c8ffb96c66a881," GitHub, [Online]. Available: https://github.com/atlassian/react-beautiful-dnd/tree/1a380855d0d008c5d6ef8e34c7c8ffb96c66a881 . [Accessed March 2023].

[32] Amzn, "GitHub - amzn/style-dictionary at 28787befb1a2c7173c64a2a4bef5b029d0799cce," GitHub, [Online]. Available: https://github.com/amzn/style-dictionary/tree/28787befb1a2c7173c64a2a4bef5b029d0799cce. [Accessed March 2023].

[33] G. Yost, "sytle-dictionary," GitHub, April 2023. [Online]. Available: https://github.com/gillianyost/style-dictionary.

[34] G. Yost, "Ability to Now Run Tests in Parallel by Gillianyost Pull Request #961," GitHub, 12 April 2023. [Online]. Available: https://github.com/amzn/style-dictionary/pull/961. [Accessed 12 April 2023].

[35] A. Alshammari, C. Morris, M. Hilton and J. Bell, "FlakeFlagger: Predicting Flakiness Without Rerunning Tests," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE),* May 2021.

[36] V. Terragni, P. Salza and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER),* pp. 69-72, 2020.