UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DENINI GABRIEL SILVA

Using Noise to Detect Test Flakiness

Recife
2022

DENINI GABRIEL SILVA

Using Noise to Detect Test Flakiness

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Software Engineering and Programming Languages
Advisor: Prof. Dr. Marcelo Bezerra d'Amorim
Co-advisor: Prof. Dr. Breno Alexandro Ferreira de Miranda

Recife
2022

Catalogação na fonte
Bibliotecária Nataly Soares Leite Moro, CRB4-1722

S586u Silva, Denini Gabriel
Using noise to detect test flakiness / Denini Gabriel Silva. – 2022.
62 f.: il., fig., tab.

Orientador: Marcelo Bezerra d'Amorim.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.
Inclui referências.

1. Engenharia de software e linguagens de programação. 2. Android. 3. Teste de software. 4. Depuração. 5. Evolução de software. I. d'Amorim, Marcelo Bezerra (orientador). II. Título

005.1 CDD (23. ed.) UFPE - CCEN 2022 – 32

Denini Gabriel Silva

“Using Noise to Detect Test Flakiness”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 25/02/2022.

BANCA EXAMINADORA

_____

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática / UFPE

_____

Prof. Dr. Márcio de Medeiros Ribeiro
Instituto de Computação / UFAL

_____

Prof. Dr. Marcelo Bezerra d'Amorim
Centro de Informática / UFPE
(**Orientador**)

This too shall pass:

There was a king who had everything in life, but he was confused. He decided to consult the wise men of the kingdom and said to them: I don't know why I feel strange and I need to have peace of mind. I need something that makes me happy when I'm sad and that makes me sad when I'm happy. The sages decided to give the king a ring, as long as the king met certain conditions. Underneath the ring there is a message but the king should only open the ring when he is in an intolerable moment, when all is lost, nothing else can be done, so the king should open the ring. The king followed the advice. One day the country went to war and lost. The kingdom was lost, but it could still be regained. He fled the kingdom to save himself. The enemy followed him, but the king rode until he lost his companions and his horse. He followed on foot, alone, and the enemies behind. His feet were bleeding, but he had to keep running. The enemy approaches and the king, almost fainting, arrives at the edge of a precipice. There is no way out, but the king thought: "I am alive, maybe the enemy will change direction, the condition is not fulfilled". Look into the abyss and see lions below, there's no other way. Enemies are catching up to him, so the king opens the ring and reads the message: "This too shall pass." Suddenly, the king relaxes. This too shall pass. And naturally the enemy changed direction. The king returns and reconquers his country. There was a great party, the people danced in the streets and the king was very happy, he cried with so much joy and suddenly he remembered the ring, opened it and read the message "This too shall pass". Again he relaxed and thus gained wisdom and peace of mind.

I dedicate this work to my family, and professors who gave me the necessary support to get here. I also dedicate this work to all those who never stopped dreaming, even if everything else does the opposite. I finish dedicating this work to the person who wrote this text that helped me a lot.

ACKNOWLEDGEMENTS

I would like to start by thanking the people who encouraged me to join the CIn: João Neto, Matheus and Professors Wylliamss and Elyda. Thank you very much for your advice.

Matheus "bought" the idea of going to do the master's with me and that contributed a lot to my decision to go. Thank you my friend!

Thanks to Keila and Irvin for their help during classes and at Labes. Thanks to my friends Geovanne and Thomás for the help even though you two are in another institution.

I thank Professor Marcelo d'Amorim for his trust in me in the beginning when he accepted me and for the countless advices that resulted in these works. It is an honor to be guided by you.

Professor Leopoldo Teixeira, for helping me since the beginning of my journey, and to Professor Breno Miranda, for helping me in the final part of this work, I am proud to be part of the STAR group.

I would like to thank the colleagues who contributed with me during this period: João Pedro, Marcello, Jordan, Stefan and Shouvick. Thank you very much for your contribution.

I give special thanks to my fiancée Assíria Campos for always being by my side, my right arm. Assíria, this achievement is also hers.

I thank the committee for the evaluation and for the significant feedback that contributed to this work.

I thank my family, especially my parents Daniel and Cida and my brother Dennys, who helped me a lot with all their support. I end by thanking God for the Grace.

ABSTRACT

A test is said to be flaky when it non-deterministically passes or fails in different runs on the same configuration (e.g., code). Test flakiness negatively affects regression testing as failure observations are not necessarily an indication of bugs in the program. Static and dynamic techniques for detecting flaky tests have been proposed in the literature but they are limited. Prior studies have shown that test flakiness is mostly caused by concurrent behavior. Based on that observation, we hypothesize that adding noise in the environment (stress tests consuming machine resources such as CPU and memory) can interfere in the ordering of program events and, consequently, it can influence the test outputs. We propose Shaker, a practical technique to detect flaky tests by comparing the outputs of multiple test runs in noisy environments. Compared with a regular test run, one test run with Shaker is slower as the environment is loaded, i.e., the process that runs a given test competes for resources with stressor tasks that Shaker creates. However, we conjecture that Shaker pays off by detecting flakiness in fewer runs compared with the alternative of running the test suite multiple times in a regular (non-noisy) environment. We evaluated Shaker using a public benchmark of flaky tests, obtaining encouraging results. For example, we found that (1) Shaker is 96% precise; it is almost as precise as ReRun, which by definition does not report false positives, that (2) Shaker's recall is much higher compared to ReRun's (95% versus 65%), and that (3) Shaker detects flaky tests much more efficiently than ReRun, despite the execution overhead associated with noise introduction. To sum up, results indicate that noise is a promising approach to detect flakiness.

Keywords: software and its engineering; android; software testing; debuggin; software evolution.

RESUMO

Um teste é dito como "flaky" quando passa ou falha de forma não determinística em diferentes execuções na mesma configuração (por exemplo, código). o teste flaky afeta negativamente o teste de regressão, pois as observações de falha não são necessariamente uma indicação de bugs no programa. Técnicas estáticas e dinâmicas para detecção de testes flaky têm sido propostas na literatura, mas são limitadas. Estudos anteriores mostraram que testes flaky são causados principalmente por comportamentos de concorrência. Com base nessa observação, levantamos a hipótese de que a adição de ruído no ambiente (testes de estresse consumindo recursos da máquina, como CPU e memória) pode interferir na ordenação dos eventos do programa e, consequentemente, pode influenciar as saídas do teste. Propomos Shaker, uma técnica prática para detectar testes flaky comparando as saídas de várias execuções de teste em ambientes ruidosos. Em comparação com uma execução de teste normal, uma execução de teste com Shaker é mais lenta à medida que o ambiente é carregado, ou seja, o processo que executa um determinado teste compete por recursos com taks de estressores que Shaker cria. No entanto, conjecturamos que Shaker compensa ao detectar falhas em menos execuções em comparação com a alternativa de executar o conjunto de testes várias vezes em um ambiente normal (sem ruído). Avaliamos Shaker usando um benchmark público de testes flaky, obtendo resultados encorajadores. Por exemplo, descobrimos que (1) Shaker é 96% preciso; é quase tão preciso quanto ReRun, que por definição não reporta falsos positivos, (2) O recall de Shaker é muito maior comparado com ReRun (95% versus .65%), e que (3) Shaker detecta testes flaky com muito mais eficiência do que ReRun, apesar da sobrecarga de execução associada à introdução de ruído. Em suma, os resultados indicam que o ruído é uma abordagem promissora para detectar testes flaky.

Palavras-chaves: software e suas engenharias; android; teste de software; depuração; evolução de software.

LIST OF FIGURES

Figure 1 – SHAKER's workflow. 22
Figure 2 – Original probability matrix M. 25
Figure 3 – Original matrix M and corresponding abstracted matrix A. 25
Figure 4 – SHAKER's GitHub Actions workflow. 27
Figure 5 – Histograms of ratio of failures for execution with and without noise. . . 35
Figure 6 – Distribution of *standard deviations of failure rates*. 35
Figure 7 – Number of flaky tests detected over time. 42
Figure 8 – Distribution of root causes and comparison with [Luo et al. 2014]. . . . 46

LIST OF TABLES

Table 1 – Causes of test flakiness reported by Luo et al. [Luo et al. 2014]. 23
Table 2 – Projects available in the ICSME2020 data. 30
Table 3 – Projects available in the FlakyAppRepo data set. 31
Table 4 – Detailed results from ANOVA for RQ2 37
Table 5 – Post-hoc analysis for RQ2. 37
Table 6 – Illustration of Shaker and ReRun's progress for the Kiss project. . . . 39
Table 7 – Precision and Recall achieved by Shaker and ReRun 40
Table 8 – Information for the 4 test cases considered as "False Positives". 41
Table 9 – Comparison of ReRun and Shaker on other Java data sets. 44
Table 10 – Scientific papers produced. 56
Table 11 – Other related publications. 56

LIST OF ABBREVIATIONS AND ACRONYMS

AVD Android Virtual Device

CI Continuous Integration

CONTENTS

1 INTRODUCTION 13
1.1 PROPOSAL 14
1.2 RESEARCH METHODOLOGY 15
1.3 RESULTS 16
1.4 OUTLINE 16

2 BACKGROUND 17
2.1 OVERVIEW 17
2.2 ANDROID CONCEPTS 18
2.3 NOISE GENERATION TOOLS 18
2.4 MOTIVATING EXAMPLES 19
2.4.1 AntennaPod Example 19
2.4.2 Paintroid Example 20

3 SHAKER: A TOOL TO DETECT FLAKY TESTS USING NOISE . 22
3.1 USING STRESS-NG FOR GENERATING NOISE 24
3.2 STEP 1: DISCOVERING CONFIGURATIONS 24
3.2.1 Random 24
3.2.2 Greedy and MHS 24
3.3 STEP 2: DISCOVERING FLAKIES 27
3.4 USING SHAKER 28
3.4.1 GitHub Action 28
3.4.2 CLI 28

4 OBJECTS OF ANALYSIS 30
4.1 THE ICSME2020 DATASET 30
4.2 THE FLAKYAPPREPO DATASET 31

5 EVALUATION 33
5.1 SETUP 33
5.2 FEASIBILITY 33
5.2.1 Answering RQ1.1 (Do tests fail more often in noisy environments than in regular (non-noisy) environments?): 33
5.2.2 Answering RQ1.2 (How repeatable is the discovery of flaky tests under a given noise configuration?): 34
5.3 COHERENCE 36

5.3.1 Answering RQ2 (Is configuration selection (Greedy or MHS) more effective than Random?): 36
5.4 PERFORMANCE . . . 38
5.4.1 Answering RQ3.1 (How many flakies does Shaker report incorrectly?): 39
5.4.2 Answering RQ3.2 (How many flakies does Shaker miss?): 41
5.4.3 Answering RQ3.3 (How quickly Shaker finds flakies?): 41

6 EFFECTIVENESS OF SHAKER ON NEW DATA SETS . . . 43
6.1 NEW DATA SET OF ANDROID PROJECTS: "ANDROID HALL OF FAME" 43
6.2 DATA SET OF NON-ANDROID PROJECTS . . . 44
6.3 DISCUSSION OF THE ROOT CAUSES OF FLAKY TESTS DETECTED BY SHAKER . . . 45
6.3.1 Examples from "Android Hall of Fame" . . . 46
6.3.2 Example from the data set of Non-Android Projects. . . . 47

7 THREATS TO VALIDITY . . . 49
7.1 INTERNAL VALIDITY . . . 49
7.2 EXTERNAL VALIDITY . . . 49
7.3 CONSTRUCT VALIDITY . . . 50
7.4 CONCLUSION VALIDITY . . . 50

8 RELATED WORKS . . . 51
8.1 EMPIRICAL STUDIES ABOUT BUGS IN TEST CODE . . . 51
8.2 TECHNIQUES TO DETECT TEST SMELLS . . . 52
8.3 TECHNIQUES TO DETECT FLAKY TESTS . . . 52
8.4 STRESS TESTING TECHNIQUES FOR DETECTION OF CONCURRENCY BUGS . . . 54

9 CONCLUSION AND FUTURE WORKS . . . 55
9.1 PUBLICATIONS . . . 56

REFERENCES . . . 57

1 INTRODUCTION

A test is said to be *flaky* when it non-deterministically passes or fails on the same configuration (e.g., machine, code, etc.) [Micco 2016]. For example, a test case of a mobile app can non-deterministically fail because of timing constraints. The test fails if a certain event does not occur within one second after the `wait` statement is executed. A possible solution that the developer could do would be to increase the waiting time, but it makes the test execution slow whereas reducing the waiting time makes the test more fragile (or flaky).

Test flakiness is a serious problem in industry. Most test failures at Google are due to flaky tests [Micco 2016, Listfield 2017]. At Microsoft, the presence of flaky tests also imposes a significant burden on developers. Lam et al. [Lam et al. 2019] reported that 58 Microsoft developers involved in a survey considered flaky tests to be the second most important reason, out of 10 reasons, for slowing down software deployments. Other organizations share similar problems [Harman and O'Hearn 2018, Developers 2021].

Three main strategies exist to deal with flaky tests: (i) prevent, (ii) detect (before running tests), and (iii) rerun (after observing failure). Prevention consists of regulating software development to prevent flakiness. For example, at Google, developers are encouraged to write single-threaded tests to avoid flakiness [Winters, Manshreck and Wright 2020]. *Prevention can be challenging*, especially when developers need to write tests beyond unit tests. Detection consists of analyzing the test cases before they are executed in regression testing. This strategy has been under active investigation in research. However, *existing detection techniques are limited*. Static detectors are imprecise [Herzig and Nagappan 2015, King et al. 2018, Pinto et al. 2020, Verdecchia et al. 2021] and dynamic detectors are limited in scope [Bell et al. 2018, Lam et al. 2019, Shi et al. 2019, Dong et al. 2020]. Finally, ReRun consists of re-executing for multiple times test cases that have failed during regression testing. A test that failed and then passed —on a fixed version of the application code— is considered flaky, and vice-versa. The status of a test that persistently failed is unknown, but developers typically treat this scenario as a problem in application code as opposed to a bug in test code. Although popular in industry [Micco 2016, Palmer 2019], *ReRun is expensive and counter-productive*. Rerunning failing tests consumes a lot of computing power.[1] Google, for example, uses 2-16% of its testing budget just to rerun flaky tests [Micco 2016]. ReRun is also counter-productive. When developers observe flaky tests during regression testing, they can choose to interrupt their activities to immediately address test flakiness or to provisionally ignore the test, postponing its repair. The decision to ignore flaky tests,[2] albeit common [Thorve, Sreshtha and Meng

[1] Google's default number of reruns for a failing test is 10 [Micco 2017].

[2] For reference, the JUnit annotation `@Ignore` tells JUnit to ignore the corresponding test.

2018], is ineffective. That practice can result in observations of even more failures during software evolution, as highlighted by Rahman and Rigby [Rahman and Rigby 2018]. Furthermore, that practice can reduce the ability of the test suite to detect bugs as it is unclear when developers would eventually revise the ignored test case to eradicate its non-determinism.

1.1 PROPOSAL

Concurrency is a very common source of test flakiness. For example, Luo et al. [Luo et al. 2014] analyzed flaky tests from Apache projects and created a list with the most prevalent sources of flakiness. The top two sources from the list alone are concurrency-related and are responsible for ∼50% of the cases of flakiness. As another evidence of the importance of concurrency on test flakiness, Google encourages developers to avoid multi-threaded code when writing test cases, with the goal of reducing flakiness observations [Winters, Manshreck and Wright 2020].

We hypothesize that non-deterministic tests fail more often in "noisy" environments, i.e., environments where machine resources (e.g., CPU and memory) are under high utilization. The rationale is that noise can interfere in the ordering of observed program events and influence on the test outputs.

Based on the observation and hypothesis above, this dissertation proposes Shaker, a black-box technique to detect flaky tests. Shaker adds noise in the execution environment with the goal of detecting flaky tests due to concurrency. A stressor task is a task responsible for generating noise in the running environment. Shaker spawns *stressor tasks* in the environment that runs the tests to provoke failures and observe discrepancies in the outputs of multiple test runs. These tasks compete with the tests for machine resources (e.g., CPU or memory).

Stressor tasks can be configured with different parameters (e.g., memory usage, cpu usage, etc.). We use the term *configuration* in this dissertation to denote the configuration of a stressor task. We empirically found that arbitrarily selecting the number of configurations and parameters of each configuration is unsatisfactory as a solution to detecting flaky tests [Silva, Teixeira and d'Amorim 2020]. In one limit, selecting only one stressor task would result in many *missed* flaky tests. In the other limit, randomly selecting lots of stressor tasks would overload the machine and defeat the purpose of *efficiently* finding flaky tests. Shaker searches for a small number of configurations to maximize flakiness detection. The process of detecting flaky tests consists of two steps. In the first *offline* step, Shaker uses a sample of tests known to be flaky to search for configurations of stressor tasks that maximize the chances of detecting those flaky tests. In the second *online* step, Shaker uses those configurations to find flaky tests in the project of interest. To search for noisy configurations in the offline (training) phase, Shaker builds a matrix relating flaky tests and stress configurations. The matrix encodes the probability

of a given test to fail on a given configuration. SHAKER uses that matrix to search for sets of configurations that reveal the highest number of flaky tests. The technique can be configured with different options for selecting configurations from the probability matrix.

1.2 RESEARCH METHODOLOGY

The main purpose of our study is to understand the usage of noise to find flaky tests in real projects. We evaluated SHAKER under three dimensions: Feasibility, Coherence, and Performance.

Feasibility checks whether the introduction of noise has an impact on the observation of failures. Coherence checks whether searching for configurations is beneficial compared with randomly choosing them. Finally, performance evaluates SHAKER considering precision (i.e., ratio of warnings reported by the tool that are in the ground truth), recall (i.e., ratio of warnings in the ground truth that are reported by the tool), and efficiency (i.e., how quickly the tool detects flakiness). We used existing datasets in our evaluation and used ReRun as our comparison baseline.

In the following, we present our research questions grouped by dimensions, and we elaborate how each research question contributes to the dimensions of our study:

FEASIBILITY:

RQ1.1. *Do tests fail more often in noisy environments than in regular (non-noisy) environments?*

RQ1.2. *How repeatable is the discovery of flaky tests under a given noise configuration?*

COHERENCE:

RQ2. *Is configuration selection (Greedy or MHS) more effective than Random?*

PERFORMANCE:

RQ3.1. *How many flakies does SHAKER report incorrectly?*

RQ3.2. *How many flakies does SHAKER miss?*

RQ3.3. *How quickly SHAKER finds flakies?*

The purpose of RQ1.1 is to evaluate if executing tests in noisy environments has the effect of making them fail more often. RQ1.2 analyzes the variance of results obtained with a given configuration of the noise generator. RQ2 evaluates the coherence of SHAKER, i.e., to evaluate whether or not the proposed approaches to configuration selection perform better than random selection. RQ3.1 focuses on **precision**. The goal of this RQ is to evaluate if tests classified as flaky by SHAKER are actually flaky. RQ3.2 focuses on **recall**.

This RQ evaluates how often SHAKER misses tests that should have been classified as flaky. Finally, RQ3.3 focuses on **efficiency**. It investigates how quickly SHAKER identifies flaky tests.

1.3 RESULTS

We observed that noise has an important impact in failure observation. Considering coherence, we found that the strategies we proposed for searching configurations of a noise generator are statistically indistinguishable, but they are both superior to random search. Considering performance, we observed that (1) SHAKER is almost as precise as ReRun, which by definition does not report false positives, (2) SHAKER's recall is significantly higher compared to ReRun's (95% versus 65%), and (3) SHAKER detects flaky tests much more efficiently than ReRun (see Figure 7).

The artifacts we produced as result of this study, including supporting scripts and the full list of projects, are publicly available in our website:

<https://star-rg.github.io/shaker>

1.4 OUTLINE

The remainder of this document is organized as follows:

Chapter 2 describes some basic concepts about tests, Android, Android testing, noise generation tool and motivating examples. Chapter 3 explains how SHAKER works and how to use it. Chapter 4 presents our methodology to find subjects and datasets to conduct the study and describes our data set. . In Chapter 5 we explain our research questions and their answers that we found through the experiments. Chapter 6 we evaluate SHAKER on different data sets of Java projects. Chapter 7 presents the threats to validity of this work and chapter 8 discusses related works. Finally, Chapter 9 concludes this dissertation and elaborates on future work.

2 BACKGROUND

In this chapter, we explain the main concepts used in our work.. Initially, in Section 2.1 we provide an overview of CI, test and test flaky. In Section 2.2 and Section 2.3, respectively, we explain about Android Concepts and Noise Generation Tools. In Section 2.4, we provide two motivating examples of our approach.

2.1 OVERVIEW

Continuous Integration(CI) is a software development practice where members of a team frequently integrate their work. Usually, each person integrates at least daily, leading to multiple integrations per day. In each integration, the CI system automates the compilation, building, and testing of software (running regression tests) to detect integration errors (bugs) as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly [Meyer 2014, Luo et al. 2014].

Regression testing is a crucial part of software development. It verifies that software changes do not break existing functionality. The result of running a regression test suite is a set of test results for the tests in the suite. The results are important for developers to take action.

If all tests pass, developers typically no longer inspect test runs. But if any test fails, developers will have to find the cause of the failure to understand if recent changes have introduced a failure in the code. An important assumption of regression testing is that test results are deterministic: an unmodified test is expected to always pass or always fail for the same code under test.

When changes cause existing tests to fail, developers should inspect the failures. There are two possible outcomes. First, if failures are caused by regressions tests, the developer must revise the application code to make the test pass.

Second, tests are broken if they fail because they no longer reflect the expected behavior of the software. Developers must fix broken tests or remove them from the test suite. In some cases, developers need to change both the application code and the test code [Daniel et al. 2009]. Repairing broken tests is time-consuming but often preferable to removing tests, since removing tests reduces a test suite's ability to detect regressions that appear in later versions of the software.

Ideally, each new test failure is due to the latest changes made by the developer, who can focus on debugging these failures. Unfortunately, some failures are not due to the latest changes, but due to unstable tests. We use the following definition by [Bell et al. 2018] for what a flaky test is: *a test that can non-deterministically pass or fail when run*

on the same version of the code.

2.2 ANDROID CONCEPTS

This section introduces Android OS concepts that are important for understanding the examples. The execution of an Android app creates a separate process including an execution thread, typically called the *main* or *UI* thread. This thread is responsible for handling events such as callbacks from UI interactions, callbacks associated with the lifecycle of components, etc. Any costly operation, such as network operations, should be offloaded to separate threads to avoid blocking the main thread, and consequently freezing the UI. Blocking the main thread for more than 5 seconds results in an Application Not Responding (ANR) error, resulting in a crash of the app.[1] For those reasons, *asynchronous operations are common in Android apps*. Worker threads, which are responsible for handling these operations, are not allowed to manipulate the UI directly; that is the responsibility of the main (UI) thread. Consequently, there must be coordination among worker threads and the main thread.

For illustration, developers often use helper methods such as `Activity.runOnUiThread()` to avoid blocking and ANR errors. This method expects a task on input and executes the task immediately if the invocation was made under the UI thread. If not, the task is posted to the event queue of the UI thread. Testing frameworks, such as Espresso [Google 2020], can create asynchronous operations by means of registering them as idling resources—operations that can have an effect on later operations in a UI test.[2]

2.3 NOISE GENERATION TOOLS

A noise generator is a tool to create artificial load in a machine. A noise generator spawn "stressor" tasks on assigned machine resources. Existing tools (e.g., sysbench [Kopytov 2020], stressant [Beaupré 2020], hardinfo [Pereira 2020], GTKStressTesting [Leinardi 2020], S-tui [Manuskin 2020], Linpack Xtreme [Badit 2020]) provide different choices for noise generation. For developing our approach (Chapter 3) we selected, among the available tools, stress-ng [King and Waterland 2020], mainly because of its popularity. The following options of stress-ng were considered:

- **–cpu** *n*. Starts *n* stressor tasks to exercise the CPU by working sequentially through different CPU stress methods, like Ackermann function and Fibonacci sequence.
- **–cpu-load** *p*. Sets the load percentage *p* for the *–cpu* command.
- **–vm** *n*. Starts *n* stressor tasks to allocate and de-allocate continuously in memory.

[1] <https://developer.android.com/training/articles/perf-anr>
[2] <https://developer.android.com/training/testing/espresso/idling-resource>

- **–vm-bytes** *p*. Sets the percentage *p* of the total memory available to use by the tasks created with option *–vm*.

For example, the command `stress-ng -cpu 2 -cpu-load 50% -vm 1 -vm-bytes 30%` configures stress-ng to run two CPU stressors with 50% load each and one virtual memory stressor using 30% of the available memory. The documentation of stress-ng can be found elsewhere.[3]

2.4 MOTIVATING EXAMPLES

In this section we present two motivational examples of our approach in real and open source projects: AntennaPod[4] and Paintroid[5].

2.4.1 AntennaPod Example

AntennaPod [AntennaPod app website 2020] is an open source podcast manager for Android supporting episode download and streaming. The AntennaPod app is implemented in Java in ∼50KLOC and contains 250 GUI tests[6] written in Espresso [Google 2020] and UIAutomator [Google 2020]. Listing 2.1 shows a simplified version of a test that checks whether a podcast episode can be played twice. The test uses the Awaitility library [Awaitility Library 2020] to handle asynchronous events, such as I/O events related to notifications of media playback. Executing the statement at line 3 turns off the continuous playback option. This command stops the app from automatically playing the next episode in the episode queue after the app finishes the playback of the current episode.

[3] project page: <https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html>
[4] project page: <https://github.com/AntennaPod/AntennaPod>
[5] project page: <https://github.com/Catrobat/Paintroid/>
[6] Revision SHA used dd5234c

Listing 2.1 – AntennaPodTest

```
1  @Test
   public void testReplayEpisodeContinuousPlaybackOff() throws Exception {
3   setContinuousPlaybackPreference(false);
    uiTestUtils.addLocalFeedData(true);
5   activityTestRule.launchActivity(new Intent());
    //Navigate to the first episode in the list of episodes and click
7   openNavDrawer();
    onDrawerItem(withText(R.string.episodes_label)).perform(click());
9   onView(isRoot()).perform(waitForView(withText(R.string.all_episodes_short_label),
        1000));
    onView(withText(R.string.all_episodes_short_label)).perform(click());
11  final List<FeedItem> episodes = DBReader.getRecentlyPublishedEpisodes(0, 10);
    Matcher<View> allEpisodesMatcher = allOf(withId(android.R.id.list),isDisplayed(),
        hasMinimumChildCount(2));
13  onView(isRoot()).perform(waitForView(allEpisodesMatcher, 1000));
    onView(allEpisodesMatcher).perform(actionOnItemAtPosition(0,clickChildViewWithId(R.
        id.secondaryActionButton)));
15  FeedMedia media = episodes.get(0).getMedia();
    Awaitility.await().atMost(1, TimeUnit.SECONDS).until( () -> media.getId() ==
        PlaybackPreferences.getCurrentlyPlayingFeedMediaId()); ... }
```

Listing 2.2 – Paintroid Test

```
@Test
public void testFullscreenPortraitOrientationChangeWithShape() {
 onToolBarView().performSelectTool(ToolType.SHAPE);
 setOrientation(SCREEN_ORIENTATION_PORTRAIT);
 onTopBarView().performOpenMoreOptions();
 onView(withText(R.string.menu_hide_menu)).perform(click());
 setOrientation(SCREEN_ORIENTATION_LANDSCAPE); pressBack();
 onToolBarView().performOpenToolOptionsView().performCloseToolOptionsView(); }
```

The statement at line 4 adds local data (e.g., podcast feeds, images, and episodes) to the app whereas the execution of the statements at lines 7–14 navigate through the GUI objects with the effect of playing the first episode in the queue. Line 16 shows an assertion based on the Awaitility library. The assertion makes the test execution to wait for at most one second until a flag is in a state indicating that the episode is being played (line 16). When the play button is pressed (line 14), the app runs a custom service in the background[7]—to load the media file from the file system and, subsequently, play the media to the user. These are typically expensive IO operations. If the machine running the test is heavy-loaded at the moment of the execution, the one-second budget may be insufficient. Consequently, the execution of the test can fail with a `ConditionTimeoutException` exception raised by the Awaitility library.

2.4.2 Paintroid Example

Paintdroid is a graphical paint editor application for Android implemented in Java in ∼25KLOC containing 250 tests. One of such tests checks whether some buttons can be clicked after changing the screen orientation. Listing 2.2 shows the test. It first selects the Shape drawing option (line 3), and then sets the screen orientation to portrait (line 4). Then, it opens a menu with a list of options (line 5), e.g., the option to save an image, the option to export an image to a file, etc. The test selects the full screen option (line 6). Then, it changes orientation to landscape (line 7), exits full screen mode (line 7), clicks on the tool options to open a menu again, and then closes it (line 8). Note that there are no assertions in the test. The goal of the test is to validate that the options remain clickable as the orientation changes from portrait to landscape. Like the previous example, this test can produce different results depending on the load of the machine. More precisely, the click on the menu item (line 6) can be performed before or after the menu is rendered on the screen (line 5). As expected, the test fails if the click is performed before the menu is rendered on the screen, throwing the exception `PerformException`.[8] Changing the screen orientation corresponds to a change in the Android configuration.[9] When a configuration change happens, Android destroys and recreates the current screen, represented by an

[7] Look for “Thread” in the `PlaybackService.java` file [AntennaPod PlaybackService.java 2020].
[8] <https://developer.android.com/reference/androidx/test/espresso/PerformException>
[9] <https://developer.android.com/guide/topics/resources/runtime-changes>

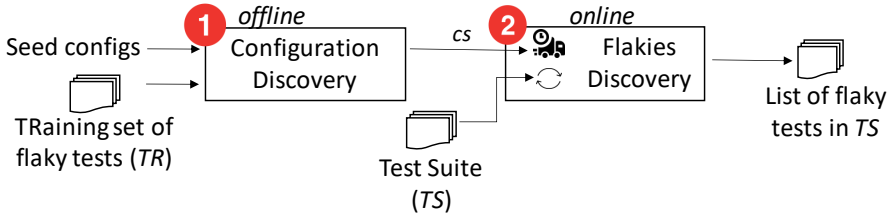
`Activity` object. This happens because changing orientation might result in a different screen layout.

3 SHAKER: A TOOL TO DETECT FLAKY TESTS USING NOISE

This chapter describes our approach, how SHAKER works, the assumptions behind its implementation, and how it is used.

Section 2.4 showed examples of flaky tests due to Async Wait. Other causes (i.e., sources) of test flakiness exist. Table 1 shows the ten different sources of flakiness catalogued by Luo et al. [Luo et al. 2014] in a study involving the analysis of 124 bug reports[1] from various projects maintained by the Apache Software Foundation. According to Luo et al., 52.2% of flaky tests are caused by the first two sources of test flakiness: Async Wait and Concurrency. These two sources are intimately related to concurrent behavior. It is worth contrasting Concurrency with Async Wait. Concurrency problems occur when the non-determinism is intrinsic in the application being tested; not the test. In test flakiness due to concurrency, the application spawns different threads, which are improperly synchronized to access the shared memory, leading to data races, atomicity violations, etc.

Figure 1 – SHAKER's workflow.



Source: Prepared by the author (2022)

SHAKER builds on the observation that test flakiness often manifests because of concurrent behavior, regardless of the source. Our hypothesis is that *(re)running tests in a noisy environment reveals more flaky tests compared to (re)running tests in a regular (non-noisy) environment*. Figure 1 shows the workflow of SHAKER. The approach consists of two steps: an offline step and an online step. In the *offline* step, SHAKER uses a sample of configurations and a sample of tests known to be flaky (TR) to search for a small set of configurations (cs) of a noise generator that optimizes the ability of test runs to reveal flakiness. In the *online* step, SHAKER uses those configurations to find flaky tests in a user-provided test suite (TS). SHAKER executes the test suite TS for a given number of times (configured) on each configuration in cs and reports a list of tests with diverging outcomes.

The following sections detail SHAKER. Section 3.1 describes how we use an off-the-shelf tool to generate noise in the execution environment. Section 3.2 describes the offline

[1] Table 2, Column "Total w. Bug Reports" [Luo et al. 2014].

Table 1 – Causes of test flakiness reported by Luo et al. [Luo et al. 2014].

Name	%	Description
Async Wait	36.4	When the test execution makes an asynchronous call and does not properly wait for the result of the call before using it.
Concurrency	15.8	When the test non-determinism is due to different threads interacting in a non-desirable manner (but not due to asynchronous calls from the Async Wait category), e.g., due to data races, atomicity violations, or deadlocks.
Test Order Dependency	9.4	When the test outcome depends on the order in which the tests are run.
Resource Leak	5.4	When the application does not properly manage (acquire or release) one or more of its resources, e.g., memory allocations or database connections, leading to intermittent test failures.
Network	4.8	When the test depends on a network resource, which is hard to control, causing flakiness.
Randomness	2.5	When the test code or application code depends random number generators without accounting for all the possible values that may be generated. For example, a test may fail only when a one-byte random number that is generated is exactly 0.
IO	2.0	IO causes other than Network listed above.
Time	2.0	When the test depends on the system time, e.g., a test may fail when the midnight changes in the UTC time zone.
Floating Point Operations	1.5	When the test performs complex floating point operations, which can produce distinct results (modulo error bounds) [Barr et al. 2013].
Unordered Collection	0.5	When iterating over unordered collections (e.g., sets), the code should not assume that the elements are returned in a particular order. If it does, the test outcome can become non-deterministic as different executions may have different orderings.
Hard to classify	19.7	-

Source: Luo et al. (2014)

step and Section 3.3 describes the online step.

3.1 USING STRESS-NG FOR GENERATING NOISE

As anticipated in Section 2.3, for developing our approach we selected the tool stress-ng [King and Waterland 2020]. We focused on CPU and memory as we empirically found that they had significant influence on the detection of test flakiness [Silva, Teixeira and d'Amorim 2020]. In addition to the stress-ng options listed in Section 2.3, SHAKER uses an option that we found important for finding flaky tests in Android apps—the number of cores available for use by an Android emulator. This option can be used to restrict an Android emulator to run on a specified number of cores. Note, however, that SHAKER is a generic method. This choice is only used for Android. The term *noise configuration* denotes an assignment of values to each of the configuration options from stress-ng.

3.2 STEP 1: DISCOVERING CONFIGURATIONS

The goal of this step is to identify noise configurations that are more likely to reveal flakiness in a test suite. This step takes as input (1) a set of *Seed configurations* and (2) a set of tests *known to be flaky*, *TR*. It reports on output a small set of configurations $cs = \{c_1, ..., c_n\}$ that are more likely to reveal flakiness. SHAKER can be configured to use one of the following search strategies: Random, Greedy, and MHS. Regardless of the chosen strategy, SHAKER runs the test suite on each given noise configuration and discards the configuration if execution exceeds *twice the original running time of the test suite*. It is worth noting that an excessively costly configuration not only slows down test execution, but also can provoke test timeouts.[2] When provoked by a noise configuration, a test timeout is an artifact of our infrastructure and it is not counted as a valid failure observation.

3.2.1 Random

The Random strategy randomly samples n configurations from their respective domains, discarding the configurations that results in long test runs. Recall that a noise generator is configured from a list of options $[o_1, ..., o_k]$, with each option $o_{1 \leq i \leq k}$ ranging over the interval lo_i-hi_i (see Section 3.1). Unlike the other search strategies, Random does not read from *TR* to steer the search.

3.2.2 Greedy and MHS

In contrast to Random, Greedy and MHS search for configurations systematically. For that, Greedy and MHS use tests that are *known to be flaky* to steer the search. Intuitively,

[2] It is not uncommon for developers to define a time limit for the execution of a test or a test suite. This is specially convenient for performance testing. In JUnit 5, a developer can write the following annotation in a test case to indicate that it should run in at most on tenth of a second `@Timeout(value=100, unit=TimeUnit.MILLISECONDS)`.

Figure 2 – Original probability matrix M.

	c_1	c_2	c_3	c_4
t_1	0.1	0.6	0.5	0.2
t_2	0.6	0.6	0.1	0.2
t_3	0.1	0.1	0.1	0.5

Source: Prepared by the author (2022)

Figure 3 – Abstracted version of matrix M with a threshold of 0.5. Original matrix M and corresponding abstracted matrix A. MHS(A)=$\{c_2, c_4\}$.

	c_1	c_2	c_3	c_4
t_1	0	1	1	0
t_2	1	1	0	0
t_3	0	0	0	1

Source: Prepared by the author (2022)

we want SHAKER to detect all these tests as flaky. We refer to this set of known flaky tests as *TR* as SHAKER uses it for training. We obtain those tests by running the test sets of different programs for multiple times, identifying the tests with non-deterministic outputs (i.e., tests that do not always pass or do not always fail). Chapter 4 explains how we obtained the training set to evaluate SHAKER. Like Random, Greedy and MHS also rely on a sample set of seed configurations drawn from their respective domains (see "Seed configs." on Figure 1). SHAKER searches for noise configuration in two steps. First, SHAKER generates a matrix encoding the probability of each sampled configuration finding flakiness in the tests in *TR*. Second, SHAKER uses that matrix to search for sets of configurations that maximize coverage of flaky tests. *Greedy and MHS differ in how they select configurations from this matrix.* The following sections elaborate each of these steps in detail.

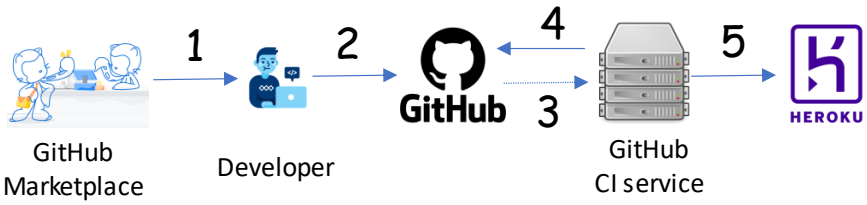
Step 1.1: Generation of probability matrix. This step takes as input a set of flaky tests *TR* and reports as output a probability matrix M, relating tests in *TR* and configurations in K, by their corresponding probabilities. The set K denotes noise configurations, randomly-sampled according to the method described above. The symbol $M[t][c]$ denotes the probability of configuration $c \in K$ detecting flakiness in $t \in TR$. To obtain approximate probability measurements, SHAKER runs each test for a user-defined number of times on each sampled configuration. The probability measurement $M[t, c]$ is obtained by dividing the number of observed failures of t by the total number of test runs in an environment with noise configuration c. Figure 2 depicts the flakiness probabilities for a scenario with three tests and four configurations.

Step 1.2: Search. The *Greedy* search strategy sorts configurations in the matrix according to their fitness value and reports the top-n configurations in the sorted list. The value n is provided by the user. We use the symbol $\mathit{fit}(c, TR)$ to denote the fitness value —a number in the 0-1 range— of a given configuration c with respect to the test set TR. The fitness value of a configuration c is obtained by computing the average probability of detecting flakiness in TR when c is used for noise generation.

Example. Consider that the set TR contains three flaky tests t_0, t_1, and t_2. Additionally, consider that a configuration c detects flakiness in TR with probabilities $[0.2, 0.5, 0.0]$. The value at index i in this list denotes the observed failure ratio of test $t_{0 \leq i \leq 2}$. The value 0.0 indicates that a test deterministically fails whereas the value 1.0 indicates that a test deterministically passes. These ratios are obtained by running the test suite for multiple times and computing the average number of failures. In this case, one test failed in 20% of the runs, another test failed in 50% of the runs, and another test did not fail at all (i.e., it deterministically passed). The fitness of configuration c is 0.23 because $\mathit{fit}(c, TR)=(0.2+0.5)/3=0.23$.

The *Minimum Hitting Set (MHS)* strategy, in contrast with the Greedy strategy, attempts to cover every test in the matrix. Recall that every test in the matrix is known to be flaky. MHS [Alon, Moshkovitz and Safra 2006] is a well-known intractable problem with efficient polynomial-time approximations [Gainer-Dewar and Vera-Licona 2016]. To sum up, MHS enables Shaker to identify minimum sets of configurations (columns of the matrix) that detect the maximum number of flaky tests (rows in the matrix). Variations of the MHS problem exist considering weights and returning complete or partial (sub-optimal) solutions [Rebert et al. 2014]. Shaker uses the unweighted and complete MHS version, which takes a *boolean* matrix as an input and produces a minimum hitting set of configurations on output. We abstracted the probability matrix M to only encode low or high likelihood of configurations detecting flaky tests. Intuitively, we are only interested in a configuration c to detect flakiness of a certain test t if the observed probability $M[t][c]$ is above a certain threshold. More precisely, Shaker computes an abstract matrix A defined as $A[t][c]=1$ if $M[t][c] >= \mathit{threshold}$, otherwise 0. Shaker runs MHS on the boolean matrix A to find a set of configurations (the columns in the matrix) that covers likely flaky tests (the rows in the matrix). Note that, although MHS assures that all flaky tests are covered (i.e., a test is covered by some configuration in the MHS), there are no guarantees that these tests will be covered in actual execution.

Figure 4 – SHAKER's GitHub Actions workflow. At step 1, the developer copies the template of the Shaker GitHub action (Listing 3.4.1), available on the GitHub Marketplace or from our web site, to update her .github/workflow/main.yml file. At step 2, the developer makes a push or pull request (configurable) to her GitHub repository. At step 3, GitHub notifies its CI service about that event. At step 4, the CI service pulls the changes from the corresponding commit from GitHub and runs the SHAKER action within a Linux container that is prepared with a tool for stressing the resources of the machine (`stress-ng`). Finally, at step 5, SHAKER notifies a web service, hosted in Heroku (could be any PaaS host), to store telemetry data about the execution.



Source: Prepared by the author (2022)

Example. Figure 3.2.2 illustrates MHS to discover configurations for detecting flakiness. For space, we used a 3x4 matrix, i.e., the test suite contains three test cases and the set of configurations includes four configurations. In practice, these matrices are much bigger. Figure 2 shows the probability matrix M, obtained with multiple executions of each test suite on each configuration. Figure 3.2.2 shows the matrix A abstracted from M. The matrix on the left shows the probabilities of each configuration detecting flakiness on TR=$\{t_1, t_2, t_3\}$. The matrix on the right-hand side is obtained using the abstraction function as described above with a threshold value of 0.5. There are five hitting sets associated with the abstract matrix, namely $\{c_1, c_2, c_4\}$, $\{c_1, c_3, c_4\}$, $\{c_2, c_3, c_4\}$, $\{c_1, c_2, c_3, c_4\}$, and $\{c_2, c_4\}$. The MHS algorithm is able to identify that the set $\{c_2, c_4\}$ is a minimal set that hits (i.e., covers) the tests in TR. In this case, it is also the minimum.

3.3 STEP 2: DISCOVERING FLAKIES

Finally, SHAKER uses the configurations obtained in Step 1 to determine which of the user-provided tests are flaky. Figure 1 illustrates the inputs and output of this step in the box named "Flakies Discovery". SHAKER runs the user-provided test suite on each configuration $c \in cs$ for a specified number of times and reports a lit of flaky tests on output when it observes differences in their test outputs. It is worth noting the tension between cost and effectiveness associated with SHAKER. The time required to run a test suite under a loaded environment should be higher compared to a regular (i.e., noiseless) execution as different tasks are competing for the machine resources. Our hypothesis is that SHAKER pays off by requiring a small number of reruns to detect flakiness.

3.4 USING SHAKER

SHAKER can be used in two ways: (1) via a GitHub Action [GitHub Actions 2022] for integration with a GitHub project or (2) via Command Line Interface (CLI). The following sections elaborate on these user modes and provide implementation details. The tool also currently supports Maven-based Java projects and `pytest`-based Python projects. Note that the tool is available with its online part, in other words it is not necessary for the developer to run the offline part (discovery of settings) and use the settings found in this work. SHAKER can be accessed at <https://star-rg.github.io/shaker>. A demo video is available at <https://youtu.be/7-aiQwOb4rA>.

3.4.1 GitHub Action

Figure 4 illustrates the workflow of SHAKER's GitHub Action. To use SHAKER's GitHub Action in a repository, the developer needs to include the contents of Listing 3.4.1 in a repository's workflow file. The effect of that inclusion is to create a new job, `shaker`, that is executed when a specified workflow is triggered. The three arguments declared on the listing are the configurable inputs from the user.[3] After finishing the workflow run, results are displayed in the Actions tab of the GitHub repository. If discrepancies in test outputs are detected, SHAKER reports the flaky tests under that tab and sends an email to the project owner.

```yaml
shaker:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
    - name: Shaker
      uses: STAR-RG/shaker@main
      with:
        tool: maven
        runs: 3
        no_stress_runs: 1
```

Listing 3.4.1: SHAKER GitHub Action configuration to be included in a YAML configuration file under `.github/workflow` directory.

The SHAKER GitHub Action relies on a Docker image, based on Ubuntu 20.04, containing all of the tools and dependencies required to run SHAKER on the cloud, i.e., from the GitHub CI service.

3.4.2 CLI

To use SHAKER through the CLI the user needs to clone the source code via `git clone https://github.com/STAR-RG/shaker`. Then, SHAKER can be invoked with the following com-

[3] Detailed explanation of these parameters can be found on the SHAKER's website.

mand `shaker/shaker.py -no-stress-runs 1 -stress-runs 4 {pytest,maven,android} directory`, where `<directory>` refers to the path of the project to be tested. The number of noisy and noiseless runs is configurable. After execution is complete, SHAKER reports whether flaky tests were found.

4 OBJECTS OF ANALYSIS

We used the following disjoint data sets of flaky tests in our evaluation: *ICSME2020* [Silva, Teixeira and d'Amorim 2020] and *FlakyAppRepo* [Dong et al. 2021]. The first data set is used for training SHAKER whereas the second is used to evaluate SHAKER.

4.1 THE ICSME2020 DATASET

In prior work [Silva, Teixeira and d'Amorim 2020], we created a data set of **74 flaky tests** from 11 Android projects mined from GitHub. We used the following search criteria to select projects:

1. the project must be written in Java or Kotlin;
2. the project must have at least 100 stars;
3. the project must include tests in Espresso or UIAutomator;
4. the project needs to be built without errors.

We sampled a total of 11 projects that satisfied this criteria and used ReRun to find test flakiness. As usual, we detected these flaky tests by re-executing test suites of the corresponding projects for 100 times using a generic Android Emulator (AVD) with Android API version 28 and observing discrepancies on the test outputs.

Table 2 – Projects available in the ICSME2020 data set and used to train SHAKER. Projects with names striked were discarded, whereas projects with names highlighted in gray color also appeared on the testing set (no overlap in flaky tests observed).

#	Name	SHA	#Stars	# Tests	#Flak	#	Name	SHA	#Stars	#Tests	#Flak
1	AntennaPod	dd5234c	4k	250	12	7	Omni-Notes	b7f9396	2.4k	10	0
2	AnyMemo	7e674fb	117	150	0	8	Orgzly	d74235e	2.1k	266	38
3	~~Espresso~~	043d028	1.1k	14	~~1~~	9	Paintroid	1f302a2	101	270	5
4	FirefoxLite	048d605	269	70	15	10	Susi	17a7031	2k	17	0
5	FlexBox-layout	611c755	17.2k	232	1	11	WiFiAnalyzer	80e0b5d	1.5k	3	0
6	Kiss	00011ce	2.1k	16	3						
Σ #Tests (#Flak)										**1,298 (74)**	

Source: Prepared by the author (2022)

Table 2 shows the projects on this data set. Column "Name" shows the name of the project, column "SHA" shows the hash of the project revision, column "#Stars" shows the number of times the project was starred on GitHub, column "#Tests" shows the number of tests cases in the project, and column "#Flak" shows the number of flaky tests we detected. The final row shows the total number of tests analyzed across all projects and

the number of flaky tests (in parentheses) in this data set. Projects with names striked were discarded because they also appeared on the testing data set with the same SHA. Projects with names in gray color also appeared on the testing set (Table 3), but there was no overlap between the training and testing sets of these projects.

4.2 THE FLAKYAPPREPO DATASET

FlakyAppRepo is a public data set[1] of flaky tests from 33 GitHub Android projects. This data set was made available by Dong et al. [Dong et al. 2021] to evaluate *FlakeScanner*, a tool for detecting flaky tests in Android apps (see Chapter 8). It is worth noting that (1) Shaker is not restricted to Android or UI tests and (2) Chapter 6 reports results of Shaker on different data sets, including non-Android projects.

Table 3 – Projects available in the FlakyAppRepo data set and used to evaluate Shaker. Projects with names striked (5 of them) couldn't be built and were discarded.

#	Name	SHA	#Stars	#Tests	#Flak	#	Name	SHA	#Stars	#Tests	#Flak
1	Amaze File Manager	eaca7050	3,7k	37	0	18	Just Weather	f7cb438bb	65	9	1
2	~~YouTube Extractor~~	53a52aa9	739	3	-	19	Kaspresso	9731d40	1,3k	72	0
3	AntennaPod	aca6e3	4k	247	43	20	KeePassDroid	33ed5c56	1,3k	59	0
4	~~Backpack Design~~	26f2441	55	248	-	21	KickMaterial	25181c9	1,6k	10	2
5	Barista	c7f16bc	1,5k	306	17	22	Kiss	d9ffdc41	2,1k	16	8
6	CameraView	68947cc	4,7k	9	0	23	MedLog	0d99aa54	0	30	4
7	Catroid	a0f2bf2	0	1,100	4	24	Minimal To Do	83bf4c6f	2,1k	7	0
8	City Localizer	db72caf	0	14	1	25	MoneyManagerEx	1ae6fd85	1l	10	0
9	ConnectBot	2b6c6e0	1,8k	46	2	26	~~My Expenses~~	d331b8b	372	75	-
10	DuckDuckGo	07d89a9	2,4k	1,234	3	27	NYBus	d14198b9	284	19	1
11	Espresso	043d028	1,1k	14	2	28	~~Omni Notes~~	eaf905c	2,4k	83	-
12	Firefox Focus	90c1e96	1,9k	72	6	29	OpenTasks	6b741e4	852	49	1
13	Firefox Lite	dcd2f44a	269	73	24	30	ownCloud	b6421e2f	3,2k	133	33
14	FlexBox-layout	d6c186b	17,2k	243	4	31	Sunflower	5829c76e1	15,2k	12	0
15	GnuCash	879596c1	1,1k	38	3	32	Surveyor	40f9448	14	46	3
16	IBS FoodAnalyzer	fe22728f	1	21	1	33	~~WordPress~~	65b5392	2,6k	106	-
17	Google I/O	4054aa3f8	21,1k	12	7						
Σ #Tests (#Flak)										**3,917 (170)**	

Source: Prepared by the author (2022)

Table 3 shows the projects on this data set. The structure of this table is as in Table 2. For the sake of reproducibility, we built each one of the 33 Android projects from this data set and re-executed its test suite for 100 times[2]. By using this method, we found a total of **170 flaky tests** — this is the set of flaky tests used as **ground truth** for evaluating the performance of Shaker (Section 5.4). It is worth noting that we were unable to compile or run the test suites of 5 projects, namely YouTube Extractor, Backpack Design, My Expenses, Omni Notes, and WordPress. The names of these projects appear crossed out

[1] <https://github.com/AndroidFlakyTest/FlakyAppRepo>

[2] We adopted the same number of repetitions (100) used by the authors of FlakyAppRepo for comparing their approach with ReRun.

on Table 3.[3] Additionally, after running the project's test suite for 100 times, no flaky tests were detected in 7 projects, namely Amaze File Manager, Camera View, Kaspresso, KeePassDroid, Minimal To Do, MoneyManagerEx, and Sunflower. Note that, although there are no guarantees that 100 repetitions is sufficient to find all flaky tests originally detected in the FlakyAppRepo data set, these seven cases provide concrete evidence of the importance of reproducing the experiments of a data set to obtain the ground truth. In total, we analyzed 3,917 tests (=4,432-3-248-75-83-106) from 28 (=33-5) projects, detecting 170 flaky tests from 21 (=28-7) projects.

[3] We contacted the authors via email seeking for help to build those apps but, as of the time of this submission, we have not received a response.

5 EVALUATION

This Chapter presents the evaluation of Shaker. For that, we considered three aspects: Feasibility, Coherence, and Performance. **Feasibility** evaluates whether adding noise consistently provokes test failures (Section 5.2). **Coherence** evaluates if the proposed methods for configuration selection (e.g., MHS) are more effective than random configuration selection (Section 5.3). Finally, **Performance** compares Shaker and ReRun considering 1) precision, i.e., the ratio of correct reports of flakiness, 2) recall, i.e., the ratio of known flakies detected, and 3) efficiency, i.e., detection time (Section 5.4).

5.1 SETUP

We performed our experiments on a machine powered by a ninth-generation Intel Core i5-7200U CPU @ 2.50GHz (base frequency), 4 cores 8 GB RAM, and with an SSD storage of 240 GB. Ubuntu version 18.04 with Linux kernel version 5.4.0-100-generic, and a generic Android Emulator (AVD) with Android API version 28 with 32 GB of storage.

5.2 FEASIBILITY

We posed the following research questions to assess the feasibility of using noise to detect flaky tests:

RQ1.1. Do tests fail more often in noisy environments than in regular (non-noisy) environments?

RQ1.2. How repeatable is the discovery of flaky tests under a given noise configuration?

The purpose of RQ1.1 is to evaluate if executing tests in noisy environments has the effect of making them fail more often. This question is important because Shaker builds on that assumption to detect flaky tests. If results are discouraging, Shaker is unlikely to be useful. RQ1.2 analyzes the variance of results obtained with a given configuration of the noise generator. Reproducibility of results is important to determine the ability of the approach to detect flakiness more deterministically.

5.2.1 Answering RQ1.1 (Do tests fail more often in noisy environments than in regular (non-noisy) environments?):

The purpose of this question is to evaluate whether or not introducing noise in the environment where tests are executed affects the rate of failures observed in test runs. This question is important as Shaker assumes that such effect exists. To answer this question, we use the ICSME2020 data set and we ran statistical tests to evaluate if there are

differences in the rate of failures observed in noisy executions versus regular, non-noisy, executions. As we evaluated the *effect* of noise on the same data set, we used a statistical test that takes two paired distributions–one distribution of numbers associated with regular executions (treatment #1) and one distribution of numbers associated with noisy executions (treatment #2). For simplicity, we fixed the noise configuration used in this experiment. We used the configuration `stress-ng -cpu 2 -cpu-load 50% -vm 2 -vm-bytes 50%`. The rationale was to maximize the load in the 4-core machine we ran this experiment. Each number in the distribution corresponds to the rate of failures in one complete execution of the test suites, i.e., the fraction of the 74 tests from our data set that manifests failures. We ran each test suite on each treatment 30 times. Consequently, each distribution contains 30 observations. We proceeded as follows to identify if there were differences in the measurements. First, we ran a Shapiro-Wilk test to check if the data were normally distributed. As we could not assume that our data were normally distributed ($W = 0.85$, $p - value < 0.05$ for the non-noisy distribution), we adopted a non-parametric statistical hypothesis test, the Kruskal-Wallis H-test. We assessed the null hypothesis (H_0) that the differences in the rate of failures are not statistically significant, i.e., introducing noise does *not* impact the rate of failure. The observed result indicated that we could reject H_0 at the 95% confidence level ($statistic = 34.55$, $p - value < 4.14e - 09$), i.e., the failure rate for noisy runs is different (higher) from that of regular runs. Since the distributions were different, we proceeded to evaluate effect size, i.e., the magnitude of the difference in the measurements. For that, we used the Vargha and Delaney $\hat{A}_{12}$ measure [Vargha and Delaney 2000], which tells us how often, on average, one technique outperforms the other. The $\hat{A}_{12}$ measure ranges from 0 to 1, and when the measure is exactly 0.5 the two techniques have equal performance. For interpreting the results, Vargha and Delaney suggest that the effect size is small if the value is over 0.56, medium if the value is above 0.64, and large if the value is over 0.71. For our data, $\hat{A}_{12}$ was 0.94, which *indicates that the effect of introducing noise is large.*

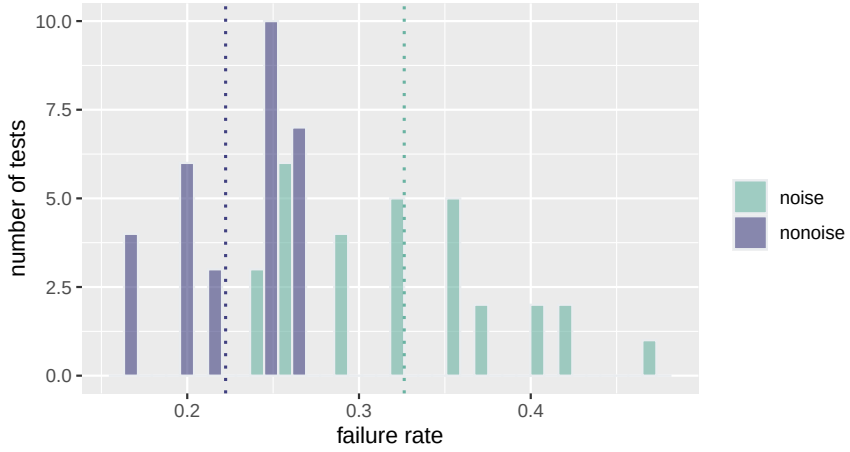
Figure 5 shows histograms of failure rates associated with noisy and noiseless test runs. Note that most executions with noisy configurations raise more failures in tests. The average rate of failures in a noisy execution is 0.33 whereas the average rate of failures in regular noiseless executions is 0.22. To sum up:

> *Summary RQ1.1:* Introducing noise in the environment increases the failure rate of test cases.

5.2.2 Answering RQ1.2 (How repeatable is the discovery of flaky tests under a given noise configuration?):

This question evaluates how repeatable are the results obtained with a given noise configuration. If results obtained with runs of the test suite with the same configuration are very

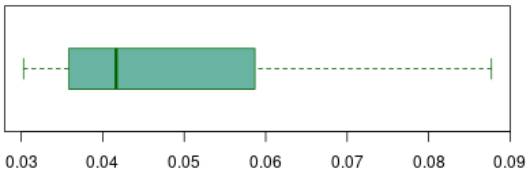
Figure 5 – Histograms of ratio of failures (X-axis) for execution with and without noise. The Y-axis shows the number of tests that manifested a rate of failures in a given interval. The vertical dotted lines indicate the average failure rate for noise (0.33) and non-noise (0.22) executions.



Source: Prepared by the author (2022)

different, then choosing configurations randomly would be no worse than searching for a set of configurations to optimize flakiness detection, as described in Section 3.2. To answer this question, we randomly selected 30 configurations and ran the test suite on each one of them for 10 times, measuring the percentage of failures detected on each execution. As a result, we obtained a different distribution of failure rates —with 10 measurements— for each one of the 30 configurations. Figure 6 shows a boxplot for the distribution of standard deviations associated with each of these 30 distributions. The average and median standard deviation was ∼.04, indicating that the average difference in measurements is small. To sum up:

Figure 6 – Distribution of *standard deviations of failure rates* showing *low variance* in results when re-executing a test suite on a given noise configuration. A data point in the distribution represents the standard deviation of one distribution of failure rates obtained with the execution of a given test suite for multiple times on a *fixed* configuration.



Source: Prepared by the author (2022)

Summary RQ1.2: The failure rates associated with the execution of a test suite –for multiple times on a fixed noise configuration– are similar.

5.3 COHERENCE

We posed the following question to assess the coherence of SHAKER, i.e., to evaluate whether or not the proposed approaches to configuration selection perform better than random selection:

RQ2. Is configuration selection (Greedy or MHS) more effective than Random?

Setup. SHAKER's configuration discovery (Step 1) requires a set of seed configurations on input. These configurations appear as columns of the concrete and abstract matrices on Figure 3.2.2. We sampled those configurations from a set of configurations uniformly distributed across the domains of the five parameters we analyzed (see Section 3.1): four parameters from stress-ng and one parameter from the AVD. We sampled a total of 30 configurations. To obtain the probabilities of the concrete matrix M, we ran each test in the training dataset (*ICSME2020*) on each of these configurations for 10 times. The result of this execution is a probability matrix with failure probability in the 0-1 range (steps of 0.1). To construct the abstract matrix A, we used a probability threshold of 0.5, i.e., values equal or above that level are set to 1 (true) and values below that level are set to 0 (false). Recall that failures provoked by timeouts are not considered (See Section 3.2).

5.3.1 Answering RQ2 (Is configuration selection (Greedy or MHS) more effective than Random?):

Recall that SHAKER selects configurations in two steps (see Section 3.2). First, it generates a probability matrix and then it selects configurations from that matrix. This research question evaluates the effectiveness of the configuration selection strategies that SHAKER uses. We evaluated three strategies for selecting configurations, namely, (i) MHS, (ii) Greedy, and (iii) Random. MHS is the technique that finds the smallest set of configurations whose execution of the test suite is capable of detecting *all* flaky tests from the training set (as per their associated probabilities in the abstract matrix). Greedy is the technique that selects configurations with maximum individual fitness scores (see Section 3.2). Random is the technique that randomly selects configurations regardless of their scores. It serves as our control in this experiment. For fairness, both Greedy and Random select the same number of configurations as MHS. The metric we used to compare techniques is the ratio of flaky tests detected from the testing set when re-running the test suite against the configurations produced by the technique. For illustration, we obtain the ratio of flaky tests for MHS as follows. Consider that we obtained four configurations with MHS. We execute the test suite four times, once for each configuration. If the test suite contains, say, 100 flaky tests and discrepancy –across these four runs– is observed in the outputs of 75 of the 100 flaky tests, the ratio will be 0.75 (=75/100).

We ran statistical tests to evaluate if there are differences in the measurements obtained by MHS, Greedy, and Random. The metric we used was the fitness score, as described above, i.e., we measured the percentage of flaky tests detected when using each selection strategy. We ran each technique for 10 times, so each distribution of measurements contains 10 samples. We ran a Shapiro-Wilk test to check if the data are normally distributed and found that the p-values are above $\alpha = 0.05$, therefore we concluded that the data are normally distributed. We then used Bartlett's test to check the homogeneity of variances, which reveals that the samples originate from populations with the same variance ($statistic = 3.52$, $p - value = 1.7e - 01$). For that, we used the one-way ANOVA (ANalysis Of VAriance) parametric test to check statistical significance of the sample means by examining the variance of the samples. The null hypothesis (H_0) is that there is no variation in the means of measurements, which would indicate that there is no impact on changing selection strategies. The test reveals that there are statistically significant differences among treatments ($statistic = 44.05$, $p - value = 9.3e - 09$). Table 4 shows the detailed results from ANOVA. Based on that observation, we performed a post-hoc paired comparison to evaluate which of the techniques differ. For that, we used Tukey's HSD test to execute multiple pairwise comparisons, one for each pair of techniques.

Table 4 – Detailed results from ANOVA for RQ2

	sum_sq	df	F	PR(>F)
C(treatments)	0.122022	2.0	44.045455	9.283040e-09
Residual	0.033244	24.0	-	-

Source: Prepared by the author (2022)

Table 5 – Post-hoc analysis for RQ2 — Multiple Comparison of Means using Tukey HSD.

Group 1	Group 2	Mean Diff.	p adj.	Lower	Upper	Reject
Greedy	MHS	0.0189	0.5368	-0.0249	0.0627	**False**
Greedy	Random	-0.1322	0.001	-0.176	-0.0884	**True**
MHS	Random	-0.1511	0.001	-0.1949	-0.1073	**True**

Source: Prepared by the author (2022)

Table 5 shows the difference in means, the adjusted p-values, and confidence levels for all possible pairs. We observe that between-group differences are significant only when comparing Random with MHS and Random with Greedy. However, statistically, results reject the hypothesis that Greedy and MHS are significantly different. To sum up:

Summary RQ2: Results indicate that there is advantage in selecting noise configurations based on their fitness scores as opposed to randomly picking them. However, there is no statistical support to claim significant differences between MHS and Greedy.

5.4 PERFORMANCE

We posed the following research questions to assess Shaker's performance:

RQ3.1. How many flakies does Shaker report incorrectly?

RQ3.2. How many flakies does Shaker miss?

RQ3.3. How quickly Shaker finds flakies?

RQ3.1 focuses on **precision**. The goal of this RQ is to evaluate if tests classified as flaky by Shaker are actually flaky. RQ3.2 focuses on **recall**. This RQ evaluates how often Shaker misses tests that should have been classified as flaky. Finally, RQ3.3 focuses on **efficiency**. It investigates how quickly Shaker identifies flaky tests.

Techniques. We compared Shaker against ReRun, a technique that re-executes a test multiple times in a noiseless environment to find discrepancies in the test outputs.[1]

Setup. To answer the questions related to performance, we use the ICSME2020 dataset (Table 2) as the training set and the FlakyAppRepo dataset (Table 3) as the testing set. We configure Shaker to use the 8 configurations obtained by MHS when answering RQ2. For comparing the performance of Shaker against ReRun we proceed as follows: i) For each subject in the training set, we run the subject's test suite with noise (Shaker) and without noise (ReRun) in a series of iterations; ii) At each iteration, Shaker runs the subject's test suite 8 times, one for each noisy configuration. For fairness, ReRun also runs the subject test suite 8 times. This way, the number of test runs for each technique is always the same at the end of each iteration; iii) We stop when Shaker and ReRun are unable to identify *new* flaky tests after a sequence of 5 iterations. To illustrate this process, Table 6 shows the actual results observed for one of the projects in our testing dataset (Kiss). Note that both techniques are able to find the same amount of flaky tests at the end of iteration #1. (We sorted the test names to facilitate illustration.) At iterations #2 and #3, Shaker revealed new flaky tests; ReRun did not reveal any. Note that ReRun misses the flaky test `testBatterySettingAppears` (highlighted in bold face in the table), which

[1] The term ReRun is often used to refer to the mechanism using during regression testing to detect test flakiness *reactively*, upon observation of test failures. Here, we use the term to refer to a technique that performs as Shaker, but re-executes tests in noiseless environments.

Table 6 – Illustration of Shaker and ReRun's progress for the Kiss project. Each row displays the list of distinct flaky tests identified at a given iteration. In this experimental setup, Shaker runs the test suite eight times on every iteration; once for each configuration that MHS reports (see Section 5.3.1). ReRun executes the test suite for the same number of times. Execution stops at iteration 15, after five consecutive iterations without new flaky tests identified by Shaker or ReRun. The test highlighted in bold was identified by Shaker at iteration 3, but missed by ReRun in the 15 iterations.

Iteration #	Shaker	ReRun
1	testExternalBarHiddenWhenViewingAllApps testInternalBarDisplayedWhenViewingAllApps testKissBarHidden testSearchResultAppears	testExternalBarHiddenWhenViewingAllApps testInternalBarDisplayedWhenViewingAllApps testKissBarHidden testKissBarEmptiesSearch
2	testCanTypeTextIntoSearchBox testKissBarEmptiesSearch	
3	**testBatterySettingAppears**	
4		testInternalBar...AllAppsWithExternalModeOn
5	testInternalBar...AllAppsWithExternalModeOn	testSearchResultAppears
6		
7		
8		
9		
10		testCanTypeTextIntoSearchBox
11		
12		
13		
14		
15		

Source: Prepared by the author (2022)

is reported by Shaker at iteration #3. Shaker saturates at iteration #5, i.e., no new flaky tests were reported by Shaker after that iteration. ReRun did not reveal anything for the next four iterations, and then, at iteration #10, it revealed again a new flaky test (a test that was reported as flaky by Shaker at iteration #2). After iteration #10, no new flaky tests were reported for the next five iterations. Execution of the experiments stops at iteration #15, when it reaches our stopping criterion. As highlighted on Table 7, it is worth noting that Shaker detects *all* flaky tests in the ground truth (Section 4.2) on this example. Recall that we do not consider failures provoked by timeouts (See Section 3.2).

Metrics. We use $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$ as metrics to answer RQ3.1 and RQ3.2, respectively. TP denotes the number of true positives, i.e., it indicates the number of correct classifications of flaky tests. FP denotes the number of false positives, i.e., it indicates the number of incorrect classifications of non-flaky tests as flaky. FN denotes the number of false negatives ("misses"), i.e., it shows the number of incorrect classifications of flaky tests as non-flaky. Precision is a proxy for the technique's correctness whereas recall is a proxy for the technique's completeness. To answer RQ3.3, we use the Area Under the Curve (AUC) denoting the cumulative number of flaky tests detected by a technique over time.

5.4.1 Answering RQ3.1 (How many flakies does Shaker report incorrectly?):

Table 7 reports precision and recall of Shaker and ReRun. Each row corresponds to one of the projects from the FlakyAppRepo data set that manifested flakiness (see Sec-

Table 7 – Precision and Recall achieved by Shaker and ReRun

		Shaker					ReRun				
Project	GT	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
AntennaPod	43	40	2	3	0.95	0.93	30	0	13	1.00	0.70
Barista	17	16	0	1	1.00	0.94	8	0	9	1.00	0.47
Catroid	4	3	0	1	1.00	0.75	1	0	3	1.00	0.25
City Localizer	1	1	0	0	1.00	1.00	1	0	0	1.00	1.00
ConnectBot	2	2	0	0	1.00	1.00	0	0	2	n/a	0.00
DuckDuckGo	3	3	0	0	1.00	1.00	1	0	2	1.00	0.33
Espresso	2	2	0	0	1.00	1.00	1	0	1	1.00	0.50
Firefox Focus	6	5	2	1	0.71	0.83	4	0	2	1.00	0.67
Firefox Lite	24	23	0	1	1.00	0.96	17	0	7	1.00	0.71
FlexBox-layout	4	3	0	1	1.00	0.75	3	0	1	1.00	0.75
GnuCash	3	3	0	0	1.00	1.00	1	0	2	1.00	0.33
IBS FoodAnalyzer	1	1	0	0	1.00	1.00	0	0	1	n/a	0.00
Google I/O	7	7	0	0	1.00	1.00	2	0	5	1.00	0.29
Just Weather	1	1	0	0	1.00	1.00	1	0	0	1.00	1.00
KickMaterial	2	2	0	0	1.00	1.00	0	0	2	n/a	0.00
Kiss	8	8	0	0	1.00	1.00	7	0	1	1.00	0.88
MedLog	4	4	0	0	1.00	1.00	2	0	2	1.00	0.50
NYBus	1	1	0	0	1.00	1.00	0	0	1	n/a	0.00
OpenTasks	1	1	0	0	1.00	1.00	1	0	0	1.00	1.00
ownCloud	33	33	0	0	1.00	1.00	29	0	4	1.00	0.88
Surveyor	3	3	0	0	1.00	1.00	1	0	2	1.00	0.33
Σ	170	162	4	8	0.98	0.95	110	0	60	1.00	0.65

Source: Prepared by the author (2022)

tion 4.2). We use *Precision* as metric to answer RQ3.1. Precision measures the fraction of tests reported as flaky that are actually flaky. Column GT (*ground truth*) contains the total number of flaky tests for a given subject. Note that, by definition, ReRun is always correct ($Precision = 1.00$ in all rows from Table 7). Overall, Shaker is very precise and achieved maximum precision for 19 out of 21 subjects for which flaky tests were detected. The overall precision of Shaker was 0.98 ($Precision = \frac{162}{162+4}$). Shaker flagged as flaky four tests for which we cannot confidently determine if they are flaky. As such, we conservatively assumed those tests are non-flaky. Note that, because we obtained our ground truth by re-executing tests a fixed number of times, it under-approximates the actual set of flaky tests. To sum up, those four tests can actually be flaky, but observing flakiness is difficult because of a small failure rate. In an additional experiment, we ran ReRun 10,000 times for these four tests, and we noticed that their failure rate is indeed very low, so these could be "hard to find" flakies. Table 8 shows the results for these possible false positives, indicating the total number of failures among the 10,000 executions, the failure

rate, and the number of executions until first failure.

Table 8 – Detailed failure information for the 4 test cases considered as "False Positives" for SHAKER

Project	Test	#Failures (10K runs)	Failure Rate	First Failure
AntennaPod	testEventsGeneratedCaseMediaDownloadSuccess_noEnqueue	4	0.04%	2653
AntennaPod	testKeepEmptyQueueSorted	5	0.05%	3410
Firefox Focus	migrationFrom1To2_containsCorrectData	5	0.05%	1582
Firefox Focus	editBookmarkWithClearingLocationContent_saveButtonIsDisabled	8	0.08%	3917

Source: Prepared by the author (2022)

Summary RQ3.1: SHAKER is very precise. When applied to the subjects in the testing set, it achieved an overall precision of 0.98. ReRun, by definition, is always correct and achieved an overall precision of 1.00.

5.4.2 Answering RQ3.2 (How many flakies does Shaker miss?):

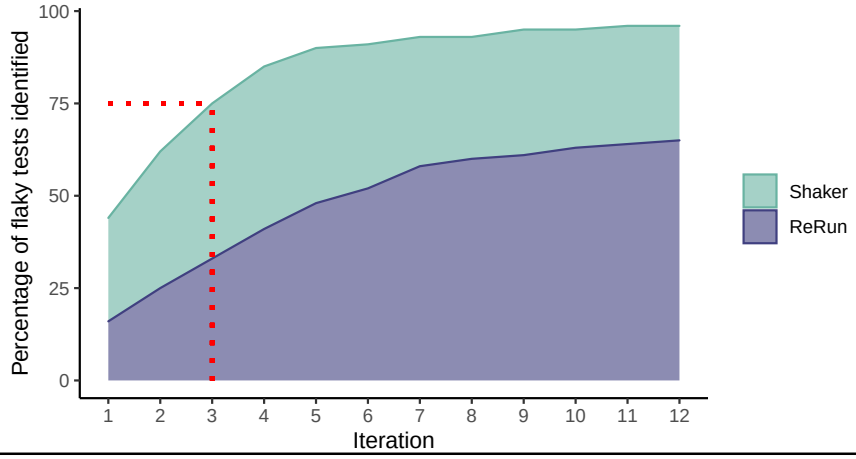
We use *recall* to answer RQ3.2. Recall measures the fraction of correct flaky classifications (i.e., reports of the technique) over the total number of flaky tests (i.e., tests that are part of the ground truth). Table 7 reports recall under columns "Recall". SHAKER achieved a recall of 0.95 ($=\frac{162}{162+8}$); it missed **8** flaky tests. ReRun achieved a recall of 0.65 ($=\frac{110}{110+60}$); it missed **60** flaky tests. Note that the number of times we ran ReRun in this experiment was limited, if we ran indefinitely it would eventually reveal the 60 tests as well, as these were discovered earlier through ReRun. SHAKER achieved maximum recall for 15 subjects (out of 21). In those cases, SHAKER correctly detected all flaky tests in the ground truth associated with the project. For ReRun, maximum recall was observed in 3 subjects only.

Summary RQ3.2: SHAKER misses far fewer flaky tests from the GT when compared with ReRun. SHAKER misses only 8 flaky tests whereas ReRun misses 60 flaky tests.

5.4.3 Answering RQ3.3 (How quickly Shaker finds flakies?):

To answer RQ3.3 we analyze the distribution of flaky tests identified by SHAKER and ReRun over time across all projects. Figure 7 displays the area plots for each technique. The y axis shows the percentage of flaky tests identified and the x axis shows the iteration number according to the description of our setup. The plots —green for SHAKER and purple for ReRun— represent the cumulative number of flaky tests identified over time, and the area that is formed between the curve and the coordinate axis, the Area Under the

Figure 7 – Number of flaky tests detected over time. Numbers highlighted match those highlighted on Table 7, column Σ.



Iteration	1	2	3	4	5	6	7	8	9	10	11	12
# flaky tests												
Shaker	75	31	22	15	10	2	3	0	2	1	1	0
ReRun	28	15	13	13	13	6	10	4	1	4	1	2
Cumulative												
Shaker	75	106	128	143	153	155	158	158	160	161	162	162
ReRun	28	43	56	69	82	88	98	102	103	107	108	110
Cumulative (%)												
Shaker	44	62	75	84	90	91	93	93	94	95	95	95
ReRun	16	25	33	41	48	52	58	60	61	63	64	65

Source: Prepared by the author (2022)

Curve (AUC),[2] is a proxy for the speed at which flaky tests are detected by a technique. As these plots show, SHAKER detects more flaky tests than ReRun in the first iterations. In the very first iteration, SHAKER identifies 75 flaky tests whereas ReRun identifies 28. Note that SHAKER not only identifies more flaky tests when compared with ReRun (as per max value in the y axis), but SHAKER identifies them faster (AUC of 82% versus AUC of 47%, respectively). As the dashed red lines show, after 3 iterations, SHAKER was able to detect 75% of all flaky tests available in the ground truth. The table on Figure 7 shows detailed progress information of techniques, showing numbers of flaky tests detected by each technique over time.

> *Summary RQ3.3:* SHAKER finds more flaky tests than ReRun and finds them faster. SHAKER's AUC is .82 in contrast to .47 of ReRun.

[2] We calculate AUC using the trapezoidal rule, a standard (and old) method to approximate the definite integral of a function. We reported the *normalized* AUC, obtained by dividing the AUC of a plot by the (ideal) AUC of the technique that detects all flaky tests in the first iteration.

6 EFFECTIVENESS OF SHAKER ON NEW DATA SETS

This Chapter evaluates Shaker on different data sets of Java projects with two goals:

1. to check whether results of Shaker on a different Android dataset are comparable to the results obtained in the previous section (Section 6.1);

2. to check the influence of using non-Android projects on the results of Shaker (Section 6.2).

6.1 NEW DATA SET OF ANDROID PROJECTS: "ANDROID HALL OF FAME"

To run this experiment, we selected a sample of Android apps from the Android "Hall of Fame" data set created by [Cruz, Abreu and Lo 2019] including 1K apps. Intuitively, projects in this data set follow good testing practices.[1] We filtered apps in this data set according to the following criteria. We selected apps written in Java or Kotlin that use GUI test cases written in the Espresso framework. This criteria resulted in 43 apps. Of these, 2 apps were used in the previous experiment, 17 apps could not be built because of broken dependencies, and 4 apps had the test suite with all tests failing. This left us with 20 apps (=43-23) to evaluate. Table 9 shows the results we obtained for this experiment under the section "Android Hall of Fame". The table only includes projects with flaky tests that Shaker or ReRun detected. In total, Shaker detected **84%** (=(52+16)/(52+13+16)) of the flaky tests detected by one of the techniques whereas ReRun detected **36%** (=(13+16)/(52+13+16)) of the flaky tests detected by one of the techniques. These results show that the difference in recall between Shaker and the baseline was even higher in this data set compared to the results reported on Section 5.4.2.

[1] <https://luiscruz.github.io/android_test_inspector/>

Table 9 – Comparison of ReRun and SHAKER on other Java data sets.

#	Name	SHA	# Stars	# Tests	# Flaky Tests		
					Only ReRun	Only SHAKER	Both
Android Hall of Fame [Cruz, Abreu and Lo 2019]							
1	Anecdote	3bedf1d	33	1	-	1	-
2	andFHEM	00c36ee	55	23	-	7	-
3	calendula	5c86212	166	13	-	-	1
4	connectbot	2d1bd0b	1.6k	10	-	6	1
5	Equate	f633bc2	54	6	-	-	2
6	Kore	b9dcbc1	399	123	5	9	2
7	MicroPinner	3095ece	36	23	2	6	-
8	open_flood	7f834ff	119	3	-	2	-
9	otp-authenticator	e7a6e4b	125	19	1	7	4
10	poet-assistant	43c667b	45	80	3	3	-
11	shoppinglist	81bc3c2	56	7	-	2	-
12	SuntimesWidget	03e1cdc	124	141	-	-	2
13	transportr	7b1866e	650	19	-	-	1
14	uhabits	d997b13	3.4k	102	2	9	3
Total	-	-	-	622	13	52	16
Non-Android projects mined from GitHub							
1	azure-iot-sdk-java	a9226a5	143	1,500	2	9	-
2	CorfuDB	b99ecff	527	948	4	4	2
3	Chronicle-Queue	bec195b	2.2K	693	2	-	2
4	exhibitor	d345d2d	1.7K	53	-	-	1
5	flow	df7a5f8	272	4,459	-	-	2
6	hbase	d50816f	3.8K	2,083	3	1	2
7	karate	09bc49e	4.3K	404	2	1	-
8	killbill	9a6f3a4	2.2K	984	-	5	3
9	mockserver	b1093ef	3K	3,524	-	1	-
10	ozone	dfd2aaf	260	1,896	2	1	-
11	ripme	351b58c	884	54	-	1	-
12	RxJava	67c1a36	43.9K	61	-	2	-
13	strimzi-kafka-bridge	eaf86fb	2.1K	232	-	1	-
Total	-	-	-	16,891	15	26	12

Source: Prepared by the author (2022)

6.2 DATA SET OF NON-ANDROID PROJECTS

It is worth noting that Android projects include mostly UI tests and there is no inherent limitation of SHAKER that prevent use in other kinds of tests. We found important to evaluate SHAKER in other kinds of projects that do not rely on UI tests, mostly. The data set used in this experiment includes Java projects selected from GitHub according to the

following criteria. We looked for projects whose issues or pull requests contained the string "flaky" and the string "test". The rationale was that projects that manifested flaky tests in the past (as per the indication of the issue or pull request) could manifest flaky tests in the future. Then, we filtered projects that use the Maven build system and that contained more than 50 test cases. We found 81 Java projects satisfying these conditions. Of these, we discarded 17 projects manifesting one of the following problems: broken dependencies, compilation times out on a 30m time budget, and all tests failed. We used the same setup as before for comparing Shaker against ReRun. Table 9 shows the results we obtained for this experiment under the section "Non-Android projects mined from GitHub". Again, the table only includes projects with flaky tests detected by Shaker or ReRun. Note that the relative difference of flaky tests that Shaker detects is not as high as in the previous experiment. In total, Shaker detected **72%** (=(26+12)/(15+26+12)) of the flaky tests detected by one of the techniques whereas ReRun detected **51%** (=(15+12)/(15+26+12)) of the flaky tests detected by one of the techniques. This suggests, perhaps as expected, that flakiness in UI tests are more common compared to non-UI tests. In the experiment with Android apps, we only executed GUI tests, which often invoke asynchronous –inherently concurrent– operations. To sum up, Shaker is specially beneficial in UI tests. Based on this, we could evaluate UI tests involving desktop or web applications in the future.

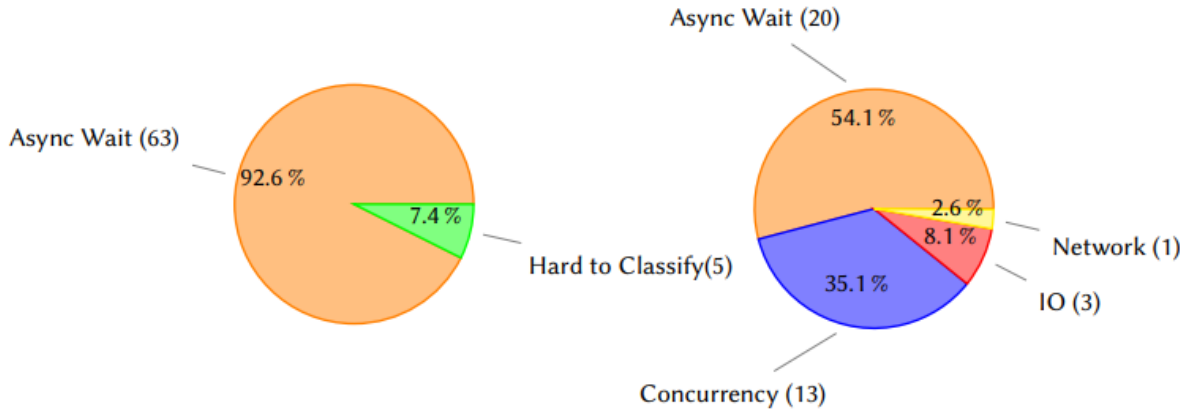
6.3 DISCUSSION OF THE ROOT CAUSES OF FLAKY TESTS DETECTED BY SHAKER

This section discusses the root causes of flakiness that we observed for the test cases that Shaker detected, focusing on the data sets discussed in this Chapter. To that end, we used the categorization proposed by Luo et al. [Luo et al. 2014] to classify the sources of test flakiness (see Table 1). To perform this classification, we analyzed the stack trace associated with test failures and the test code. The stack trace was useful to understand the test failures, and initiate the search for the root cause of flakiness. For instance, many tests in Android fail due to a `NoMatchingViewException` that is triggered when the screen does not (yet) show a particular UI widget that is expected by the test, an indication of an Async Wait problem. We also considered the application code, when needed, to understand failures by tracing the calls made by the test and what exactly was triggering the failure. Each case of flakiness was separately analyzed by two authors of this paper. For the cases the authors did not agree, all authors were involved in the discussion until reaching consensus.

Figure 8 shows pie charts with the distributions of root causes for our data sets of Android projects (Figure 8a) and Non-Android projects (Figure 8b). The undetected sources were not listed. Tables 8c relates those distributions with the distribution observed in the study conducted by [Luo et al. 2014]. Results show that –as in the study of [Luo et al. 2014]– Concurrency and Async Wait are the most common sources of flakiness.

Considering the Android data set, if we discard the flaky tests considered hard to classify, Async Wait is the only source of flakiness (92.6% of the total); no other root cause have been observed. These numbers support our expectations that SHAKER induce flakiness caused by non-deterministic behavior in concurrent programs. Note, however, that it is still possible to capture flakiness related to other sources. However, detecting flakiness not caused by concurrent behavior is not the intent of SHAKER. The following sections discuss examples of flakiness observed on these data sets.

Figure 8 – Distribution of root causes and comparison with [Luo et al. 2014].



(a) Android projects.

(b) Non-Android projects.

Category	Android	Non-Android	Luo et al. [48]
Async Wait	92.6	54.1	36.4
Concurrency	-	35.1	15.8
IO	-	8.1	2.0
Network	-	2.6	4.8
Resource Leak	-	-	5.4
Time	-	-	2.0
Hard to classify	7.4	-	19.7

(c) Prevalence of sources of flakiness from different data sets.

Source: Prepared by the author (2022)

6.3.1 Examples from "Android Hall of Fame"

Figure 8 breaks down the root causes of flakiness we observed. Note that in this data set we could only classify confidently flakiness due to async wait. We discuss a sample of examples in the following. [**Async Wait**] `otp-authenticator` is a two-factor authentication app that performs QR code reading operations using the device camera. One of its tests, `test001InvalidQRCode` [Test 001InvalidQRCode 2021], reads a QR code and then tries to perform an assertion over a UI widget that is absent from the screen at the time of running the assertion. As result, the test triggers `NoMatchingViewException`. This pattern of trying to match some UI widget that was supposed to be visible on the screen was a

common source of flakiness among Android apps, amounting to 62.9% (39/62) of the cases for this category. [Romano et al. 2021] also found that accessing UI resources before it is rendered on the screeen is a frequent reason for async wait flakiness [Dataset of "An Empirical Analysis of UI-based Flaky Tests" 2021]. It is worth noting that although Espresso provides a mechanism called "idling resources"[2] to deal with such situations, in our analysis, we found that the most prevalent repair for flaky tests in this category consisted of using `Thread.sleep`. As another example, consider CALENDULA, a personal medication management app. The test `testCreateMedicine` [Test CreateMedicine 2021] exercises the UI with the goal of saving a medicine with the name "aspirin" into the database. The setup of the test creates a fresh database. After filling the form and clicking the save buttom, the test queries the database to confirm that the medicine with the name "aspirin" was indeed saved. The test asserts that the result set returned by the query contains records. However, this assertion fails in some test runs. In Android, IO operations, such as accessing a database, need to be performed outside the main thread as to avoid blocking the UI. This test spawns a worker thread to add the record to the database and, sometimes, executes the assertion before the record appears on the database, resulting in a failure. [**Concurrency, IO, Network, Time and Resource Leak**] As Figure 8 shows, we found no instances of test flakiness caused by concurrency, IO, network, time and resource leak(age) in the Android data set, which are other common sources of flakiness observed by [Luo et al. 2014].

6.3.2 Example from the data set of Non-Android Projects.

We observed flakiness in this data set from a variety of sources (see Figure 8b), supporting our claim that various root causes listed by [Luo et al. 2014] are related to concurrency. [**Async Wait**] The Vaadin Flow [Vaadin Flow 2021] framework for web development contains another example of async wait flaky tests. The test `vaadinServlet_forDifferentRequests_shouldHaveCorrectResponse` [Test vaadinServlet 2021] makes various HTTP requests and compares the responses with corresponding expectations. In some executions, a `BindException` is triggered when preparing the HTTP server for using a particular port. The exception indicates that the address requested is already in use. The problem occurs despite the fact that there is a `teardown` method associated with the tests for releasing the resource (in this case, a network port). The non-deterministic behavior occurs because the setup method of the failing test did not wait for the release of the resource by the previously-executed test before requesting the resource. [**Concurrency**] `azure-iot-sdk-java` [Azure repository 2021] is a Java SDK for connecting devices to Microsoft's Azure IoT services. The test `authenticateWithProvisioningServiceWithX509Succeeds` [Test authenticateWithProvisioning 2021] uses the JMockit library [JMockit 2021] to mock objects from an API for testing message passing through the MQTT protocol for IoT messaging. Investi-

[2] <https://developer.android.com/training/testing/espresso/idling-resource>

gating the issue by analyzing the test code and stack trace, we discovered that the version of the library that was used introduced a race condition that made tests fail intermittently.[3] [**Network**] The test `testImagefapAlbums` [Test ImagefapAlbums 2021] downloads images from a given website. In some executions, the access to the page is slow, triggering a `SocketTimeoutException`. We found that this test was indeed tagged as flaky in their codebase. [**IO**] Apache HBase [Apache HBase 2021] is a distributed, scalable, big data store. The test `testRetrieveFromFile` [Test RetrieveFromFile 2021] performs a series of operations related to testing the behaviour of caching. It starts by initializing an object that is used as a block cache (`BucketCache`). It then adds blocks to this object, persists the cache to the file, and then checks if it can restore the cache from the file. Later it deletes the bucket cache file, restarts the object, and asserts that it is unable to restore the cache from the file. After that, it tries to delete the persistence file, but the assertion that the file exists and was correctly deleted fails because the file did not exist yet during execution. The `shutdown()` method of this object shuts down the cache and persists the information into a persistence file. [**Resource Leak and Time**] We found no instances of test flakiness caused by resource leak(age) and time issues in the Java data set.

[3] <https://github.com/jmockit/jmockit1/issues/263>

7 THREATS TO VALIDITY

Beyond our best efforts to ensure an accurate design and execution of the evaluation, our results might still suffer from validity threats. This chapter presents a summary of the potential threats to validity of our study, including threats to internal, external, construct, and conclusion validity [Wohlin et al. 2012].

7.1 INTERNAL VALIDITY

Threats to the *internal validity* compromise our confidence in establishing a relationship between the independent and dependent variables. To mitigate this threat, we carefully inspected our implementation and the results we obtained in our evaluation. In addition, we ran our experiments in different machines and different data sets to confirm the impact of noise in detecting flakiness. Another possible threat is the number of repetitions used for building our ground truth, as 100 reruns might not be enough to reveal low-probability flaky tests, i.e., tests that rarely non-deterministically fail (or pass). Prevention to this threat can be achieved by increasing the number of repetitions. However, for our experiments reported in Chapter 5, we used Android applications with GUI test suites, and each repetition is very expensive — the test suite of `ownCloud`, for example, contains 133 test cases and it takes approximately 6 minutes to run in our experimental environment; 100 repetitions takes 10 hours for this single app. While we find the choice of number of repetitions to be important to determine the ground truth, we do not consider it as a critical threat to validity as (1) we use the same ground truth to compare the two techniques (Shaker and ReRun), so we expect the two techniques to be affected in the same way and (2) the problem of determining the number of repetitions would still exist if we decide to increase its current value.

7.2 EXTERNAL VALIDITY

Threats to the *external validity* relate to the ability to generalize our results. We cannot claim generalization of our results beyond the particular set of projects studied. Our findings are limited by the projects we studied, as well as their domains. To minimize this threat, we explore two large data sets of programs. In addition to the data set of Android applications we analyzed in our previous work [Silva, Teixeira and d'Amorim 2020], we considered a new data set of Android applications for our study, the FlakyAppRepo [Dong et al. 2021]. Also, for the results reported in Chapter 6 we considered a data set of Java programs, and showed that Shaker can detect flakiness in tests that do not interact with the UI.

7.3 CONSTRUCT VALIDITY

Threats to the *construct validity* are related to the appropriateness of the evaluation metrics we used. We used popular metrics previously used in the testing literature. For example, to evaluate performance, we used precision (i.e., fraction of reports of flaky tests which are actually flaky), recall (i.e., fraction of actually flaky tests which are reported), and detection speed, as measured by the Area Under the Curve (AUC).

7.4 CONCLUSION VALIDITY

Threats to *conclusion validity* are concerned with the relationship between the treatments and the outcome of the experiments. One possible threat could be the violation of the assumptions of the statistical methods used. This threat affects our RQs differently as different statistical methods were adopted. For the case of RQ1, we consider such a threat to be low because we used non-parametric tests, which do not rely on assumptions about the distribution of the underlying data. For the case of RQ2, a parametric test was used and we controlled this threat by using a classical test that has been shown to be robust to the violation of (at least some of) their assumptions.

Another possible threat happens when a hypothesis turns out significant by chance, even when there is no difference between the treatments. Although the significant results observed in our evaluation are in line with the expectations, further studies using larger sample sizes should be considered to minimize this threat.

8 RELATED WORKS

In this chapter, we describe some related works. We organized it in four groups: Empirical studies about bugs in test code, techniques to detect test smells, techniques to detect flaky tests and stress testing techniques for detection of concurrency bugs.

8.1 EMPIRICAL STUDIES ABOUT BUGS IN TEST CODE

Different empirical studies [Luo et al. 2014, Thorve, Sreshtha and Meng 2018, Vahabzadeh, Fard and Mesbah 2015, Waterloo, Person and Elbaum 2015, Tran et al. 2019, Eck et al. 2019, Lam et al. 2020] attempted to characterize the causes and symptoms of buggy tests, i.e., tests that fail not because of a problem in the application but because of a problem in the test itself. This dissertation focuses on test flakiness, which is one important–and very common–type of test code issue. Considering test flakiness, [Luo et al. 2014] were the first to characterize the sources of test flakiness and common repairs to mitigate them. They analyzed the commit history of the Apache Software Foundation central repository looking for flakiness manifestations. They analyzed 1,129 commits including the keyword "flak" or "intermit", and then manually inspected each commit. They proposed 10 categories of root causes for flakiness and summarized the most common strategies to repair them. The majority of the problems they reported were related to timing constraints that could, in principle, be captured by Shaker. [Vahabzadeh, Fard and Mesbah 2015] mined the JIRA bug repository and the version control system of the Apache Software Foundation and found that 5,556 unique bug fixes exclusively affected test code. They manually examined a sample of 499 buggy tests and found that 21% of these false alarms were related to flaky tests, which they further classified into Asynchronous Wait, Race Condition, and Concurrency Bugs. Thorve et al. [Thorve, Sreshtha and Meng 2018] conducted a study about test flakiness focused on Android apps and observed that the causes of flakiness in Android apps are similar to those found by Luo et al. [Luo et al. 2014]. [Eck et al. 2019] conducted an experiment with 21 Mozilla developers where they asked developers to classify 200 flaky tests that these developers fixed. Once again, Concurrency and Async Wait were the most prevalent categories of flaky tests. In principle, Shaker could capture problems related to all these categories. Although we used Java programs, the technique and tool are programming-language agnostic.

More recently, [Lam et al. 2020] conducted a longitudinal study of test flakiness involving 55 Java projects to determine when flaky tests become flaky and what changes cause them to become flaky. They found that 75% of the cases of flakiness could be detected when the test is created and another 10% of the cases could be detected when the test is modified. The findings of this study suggests a method to reduce the cost of flakiness de-

tector tools, including SHAKER. More precisely, it shows a method to aggressively reduce the number of tests that require attention of flakiness detectors.

8.2 TECHNIQUES TO DETECT TEST SMELLS

Code smells are syntactical symptoms of poor design that could result in a variety of problems. Test smells manifest in test code as opposed to application code. Van Deursen et al. [Deursen et al. 2001] described 11 sources of test smells and suggested corresponding refactorings to circumvent them. More recent studies have been conducted on the same topic. Bavota et al. [Bavota et al. 2012] and Tufano et al. [Tufano et al. 2016] separately studied the sources of test smells as defined by Van Deursen et al. [Deursen et al. 2001]. They used simple syntactical patterns to detect these smells in code and then manually inspected them for validation. Bavota et al. found that up to 82% of the 637 test classes they analyzed contains at least one test smell. In related work, Tufano et al. studied the life cycle of test smells and concluded that they are introduced since test creation—instead of during evolution—and they survive through thousands of commits. Waterloo et al. [Waterloo, Person and Elbaum 2015] developed a set of (anti-)patterns to pinpoint problematic test code. They performed a study using 12 open source projects to assess the validity of those patterns. Garousi et al. [Garousi, Kucuk and Felderer 2018] prepared a comprehensive catalogue of test smells and a summary of guidelines and tools to deal with them.

Test flakiness *may* relate to test smells. For example, the use of sleeps are good predictors of flakiness according to [Pinto et al. 2020] and [Palomba, Zaidman and Lucia 2018]; they induce time constraints that could be violated in overloaded environments. We remain to investigate how test smells can help improve the effectiveness of static flakiness detectors and how detectors can be used in tandem with SHAKER.

8.3 TECHNIQUES TO DETECT FLAKY TESTS

Ideally, a test case should produce the same results regardless of the order it is executed in a test suite [Zhang et al. 2014]. Unfortunately, this is not always the case as the application code that is reached by the test cases can inadvertently modify the static area and resetting the static area after the execution of a given test is impractical. Test dependency is one particular source of test flakiness [Luo et al. 2014]. Gambi et al. [Gambi, Bell and Zeller 2018] proposed a practical approach, based on flow analysis and iterative testing, to detect flakiness due to broken test dependencies. Shi et al. [Shi et al. 2019] proposed iFixFlakies to find and fix flaky tests caused by broken test dependencies. SHAKER is complementary to techniques for capturing broken test dependencies. It remains to investigate how a technique that forcefully modifies the test orderings (e.g., discarding tests from test runs

and modifying orderings of test execution) compares with the approach proposed by Gambi et al..

Bell et al. proposed DeFlaker [Bell et al. 2018], a dynamic technique that uses test coverage to detect flakiness during software evolution. DeFlaker observes the latest code changes and marks any new failing test that did *not* execute changed code as flaky tests. The expectation is that a test that passed in the previous execution and did not execute changed code should still pass. When that does not happen, DeFlaker assumes that the changes in the coverage profile must have been caused by non-determinism. Note that DeFlaker is unable to determine flakiness if the coverage profile was impacted by the changes and it does not provoke flakiness as Shaker does. DeFlaker focuses on flakiness that could be detected during evolution, but Shaker was not developed for that purpose, thus we do not compare Shaker with DeFlaker in our evaluation. Recently, Dong et al. [Dong et al. 2020] proposed FlakeScanner (previously FlakeShovel), a tool to detect flakiness in Android apps by monitoring and manipulating thread executions to change event orderings. It directly interacts with the Android runtime, instead of generating stress loads as we did. We do not compare Shaker with FlakeScanner in our evaluation because FlakeScanner focuses on Android apps (there are different requirements and limitations compared to our approach), whereas Shaker focuses on concurrency-related issues in any kind of program (in Chapter 6 we reported the results of our evaluation with non-Android programs).

Purely static approaches have also been proposed to identify flaky tests [Herzig and Nagappan 2015, King et al. 2018, Verdecchia et al. 2021, Pinto et al. 2020]. An important benefit of these approaches is scalability. For example, it is possible to build services to proactively search for suspicious tests in open source repositories. On the downside, they only offer estimates of flakiness; re-execution is still necessary to confirm the issue. Herzig and Nagappan [Herzig and Nagappan 2015] developed a machine learning approach that mines association rules among individual test steps in tens of millions of false test alarms. Lam et al. [King et al. 2018] used Bayesian networks for flakiness classification. [Pinto et al. 2020] used binary text classification (e.g., Random Forests) to predict test flakiness. They used typical NLP techniques to classify flaky test cases—they tokenized the body of test cases, discarded stop words, converted words in camel case, and built language models from the words associated with flaky and non-flaky tests. [Alshammari et al. 2021] recently proposed the use of an additional set of features that improved the performance of classification models. Shaker is complementary to static techniques. We remain to evaluate how static classification techniques could be used in tandem with our approach. For example, by using the output of static detectors to select what tests should be re-executed in noisy environments.

While several other techniques for detecting flaky tests exist, in contrast to Shaker, they focus on some particular domain or application context, making the comparison with

SHAKER unrealistic and unfair. SHAKER is essentially ReRun running in a noisy environment. As such, the best comparison baseline — to highlight the impact of introducing noise — is really ReRun.

8.4 STRESS TESTING TECHNIQUES FOR DETECTION OF CONCURRENCY BUGS

Various techniques to stress test concurrent applications exist [Edelstein et al. 2003,Farchi, Nir and Ur 2003, Sen 2007, Bielik, Raychev and Vechev 2015]. Purely random approaches stress test the application by running multiple concurrent tests simultaneously, suspending threads, inserting random sleeps, and changing thread policies. For example, RAPOS [Sen 2007] introduces random sleeps at thread synchronization points to find data races. There are techniques that sample the space of thread interleaving more systematically with the goal of providing stronger guarantees of finding concurrency bugs [Burckhardt et al. 2010, Bielik, Raychev and Vechev 2015]. For example, Bielik et al. [Bielik, Raychev and Vechev 2015] proposes algorithms (i) to efficiently compute a Happens-Before event graph and (ii) to efficiently query that graph for potential races. Furthermore, it proposes filtering rules to reduce the number of reports (and improve the precision of the approach).

The solution of [Bielik, Raychev and Vechev 2015] is specific to Android. Their approach instruments the framework and apks to obtain event traces from which the Happens-Before graph is obtained. As such, [Bielik, Raychev and Vechev 2015] is more similar to the work of [Dong et al. 2021] than it is to SHAKER. Conceptually, stress testing technique for detection of concurrency bugs have different assumptions and provide different guarantees compared to SHAKER. For example, SHAKER requires no instrumentation or system configuration; it works out-of-the-box as it builds on an off-the-shelf noise generator.

9 CONCLUSION AND FUTURE WORKS

Flaky tests are a huge problem in industry. Their presence makes it difficult for developers to unambiguously interpret the results of failing tests during a regression testing cycle. The standard approach in industry to deal with flaky tests is to re-execute tests upon failures observed during regression. A test that fails during regression testing is considered to be flaky when it is possible to observe differences in the test outputs across multiple re-executions. This practice, however, is considered unproductive as developers need to either stop their activities to debug the flaky test or to ignore the flaky test that could be actually useful to reveal failures. Techniques have been proposed to *proactively* detect flaky tests, i.e., to detect flakiness before they are observed during regression testing. However, existing techniques are restricted to specific contexts and application scenarios. For example, various techniques have been proposed to detect flakiness due to test-order dependency.

This dissertation proposes Shaker, a lightweight technique to detect flaky tests caused by concurrency issues (e.g., async wait), which is the most prevalent source of test flakiness according to prior studies. Shaker adds noise in the environment where tests will be executed with the goal of observing different test outputs. The assumption is that noise can induce different thread interleavings in the execution of tests of concurrent programs. To that end, Shaker selects different configurations of a noise generator (e.g., `stress-ng`) to add stressor tasks that compete for machine resources (e.g., CPU, memory, etc.).

We evaluated Shaker using public benchmarks of flaky tests for Android applications, standard performance metrics (e.g., precision and recall), and using ReRun as a comparison baseline. ReRun is a technique that detects flaky tests with repeated executions of a test suite in noiseless environments. Results are encouraging. For example, we found that (1) Shaker is 96% precise; it is almost as precise as ReRun, which, by definition, does not report false positives, that (2) Shaker's recall is much higher compared to ReRun's (95% versus 65%), and that (3) Shaker detects flaky tests much more efficiently than ReRun, despite the execution overhead associated with noise introduction. Furthermore, we evaluated Shaker in other benchmarks and observed that Shaker is also highly effective in non-Android programs whose test suites are not focused on the UI.

In the future, we plan to investigate (1) how to improve Shaker's efficiency by interrupting stressor tasks during the execution of single-threaded tests based on the work of [Winters, Manshreck and Wright 2020], (2) how to improve Shaker's recall by targeting it to the tests, e.g., by running short-lived stressors on one specific machine resource when observing an specific event during the execution of a test and (3) evaluate Shaker in other scenarios such as applications in other languages or UI tests involving desktop

or web applications. SHAKER is publicly available:

Artifacts: <https://github.com/STAR-RG/shaker-artifacts-tosem>

Tool: <https://star-rg.github.io/shaker>

9.1 PUBLICATIONS

The Table 10 presents the scientific papers produced in scope of this dissertation. The Table 11 shows others publications produced during the dissertation production.

Table 10 – Scientific papers produced.

#	Reference	Type	Status
1	**Silva, D.**, Teixeira, L., & d'Amorim, M. (2020, September). Shake it! detecting flaky tests caused by concurrency with shaker. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 301-311). IEEE.	Conference	Published
2	Cordeiro, M., **Silva, D.**, Teixeira, L., Miranda, B., & d'Amorim, M. (2021, November). Shaker: a Tool for Detecting More Flaky Tests Faster. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1281-1285). IEEE.	Conference	Published
3	**Silva, D.**, Miranda, B., Teixeira, L., & d'Amorim, M. (2022, February). Using Noise to Detect Test Flakiness. Transactions on Software Engineering and Methodology (TOSEM). ACM.	Journal	Under Review

Source: Prepared by the author (2022)

Table 11 – Other related publications.

#	Reference	Type	Status
1	Mondal, S., **Silva, D.** & d'Amorim, M. (2021, September). Soundy Automated Parallelization of Test Execution. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 309-319). IEEE..	Conference	Published
2	Henkel, J., **Silva, D.**, Teixeira, L., d'Amorim, M., & Reps, T.. (2021, May). Shipwright: A Human-in-the-Loop System for Dockerfile Repair. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 1148-1160). IEEE.	Conference	Published

Source: Prepared by the author (2022)

REFERENCES

ALON, N.; MOSHKOVITZ, D.; SAFRA, S. Algorithmic construction of sets for k-restrictions. *ACM Trans. Algorithms*, Association for Computing Machinery, New York, NY, USA, v. 2, n. 2, p. 153–177, Apr. 2006. ISSN 1549-6325. Available at: <https://doi.org/10.1145/1150334.1150336>.

ALSHAMMARI, A.; MORRIS, C.; HILTON, M.; BELL, J. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In: *Proceedings of the International Conference on Software Engineering*. [S.l.: s.n.], 2021. To Appear.

ANTENNAPOD app website. 2020. <https://antennapod.org>.

ANTENNAPOD PlaybackService.java. 2020. <https://github.com/AntennaPod/AntennaPod/blob/44f35cb1f19443634cc921a1addc6f500a9a3850/core/src/main/java/de/danoeh/antennapod/core/service/playback/PlaybackService.java>.

APACHE HBase. 2021. <https://github.com/apache/hbase>.

AWAITILITY Library. 2020. <http://www.awaitility.org>.

AZURE repository. 2021. <https://github.com/Azure/azure-iot-sdk-java/>.

BADIT, E. *Linpack Xtreme*. 2020. Https://www.ngohq.com/linpack-xtreme.html. [Online; accessed Dec-2020].

BARR, E. T.; VO, T.; LE, V.; SU, Z. Automatic detection of floating-point exceptions. In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2013. (POPL '13), p. 549–560. ISBN 9781450318327. Available at: <https://doi.org/10.1145/2429069.2429133>.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: IEEE. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. [S.l.], 2012. p. 56–65.

BEAUPRé, A. *stressant*. 2020. Https://stressant.readthedocs.io/en/latest/. [Online; accessed Dec-2020].

BELL, J.; LEGUNSEN, O.; HILTON, M.; ELOUSSI, L.; YUNG, T.; MARINOV, D. Deflaker: automatically detecting flaky tests. In: ACM. *Proceedings of the 40th International Conference on Software Engineering*. [S.l.], 2018. p. 433–444.

BIELIK, P.; RAYCHEV, V.; VECHEV, M. Scalable race detection for android applications. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: Association for Computing Machinery, 2015. (OOPSLA 2015), p. 332–348. ISBN 9781450336895. Available at: <https://doi.org/10.1145/2814270.2814303>.

BURCKHARDT, S.; KOTHARI, P.; MUSUVATHI, M.; NAGARAKATTE, S. A randomized scheduler with probabilistic guarantees of finding bugs. In: *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2010. (ASPLOS XV), p. 167–178. ISBN 9781605588391. Available at: <https://doi.org/10.1145/1736020.1736040>.

CRUZ, L.; ABREU, R.; LO, D. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering*, Springer, v. 24, n. 4, p. 2438–2468, 2019.

DANIEL, B.; JAGANNATH, V.; DIG, D.; MARINOV, D. Reassert: Suggesting repairs for broken unit tests. In: IEEE. *2009 IEEE/ACM International Conference on Automated Software Engineering*. [S.l.], 2009. p. 433–444.

DATASET of "An Empirical Analysis of UI-based Flaky Tests". 2021. Https://ui-flaky-test.github.io/.

DEURSEN, A. V.; MOONEN, L.; BERGH, A. V. D.; KOK, G. Refactoring test code. In: *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. [S.l.: s.n.], 2001. p. 92–95.

DEVELOPERS, C. *Flakiness dashboard howto*. 2021. Http://www.chromium.org/developers/testing/flakiness-dashboard. [Online; accessed April-2020].

DONG, Z.; TIWARI, A.; YU, X. L.; ROYCHOUDHURY, A. Concurrency-related flaky test detection in android apps. *ArXiv*, abs/2005.10762, 2020.

DONG, Z.; TIWARI, A.; YU, X. L.; ROYCHOUDHURY, A. 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*. New York, NY, USA: Association for Computing Machinery, 2021. (ESEC/FSE 2021), p. 367–378. ISBN 9781450385626. Available at: <https://doi.org/10.1145/3468264.3468584>.

ECK, M.; PALOMBA, F.; CASTELLUCCIO, M.; BACCHELLI, A. 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*. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 830–840. ISBN 9781450355728. Available at: <https://doi.org/10.1145/3338906.3338945>.

EDELSTEIN, O.; FARCHI, E.; GOLDIN, E.; NIR, Y.; RATSABY, G.; UR, S. Framework for testing multi-threaded java programs. *Concurr. Comput. Pract. Exp.*, v. 15, n. 3-5, p. 485–499, 2003. Available at: <https://doi.org/10.1002/cpe.654>.

FARCHI, E.; NIR, Y.; UR, S. Concurrent bug patterns and how to test them. In: *Proceedings International Parallel and Distributed Processing Symposium*. [S.l.: s.n.], 2003. p. 7 pp.–.

GAINER-DEWAR, A.; VERA-LICONA, P. The minimal hitting set generation problem: algorithms and computation. *CoRR*, abs/1601.02939, 2016. Available at: <http://arxiv.org/abs/1601.02939>.

GAMBI, A.; BELL, J.; ZELLER, A. Practical test dependency detection. In: IEEE. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.], 2018. p. 1–11.

GAROUSI, V.; KUCUK, B.; FELDERER, M. What we know about smells in software test code. *IEEE Software*, IEEE, v. 36, n. 3, p. 61–73, 2018.

GITHUB Actions. 2022. <https://docs.github.com/en/actions>.

GOOGLE. *Espresso API webpage*. 2020. <https://developer.android.com/training/testing/espresso>.

GOOGLE. *UIAutomator API webpage*. 2020. <https://developer.android.com/training/testing/ui-automator>.

HARMAN, M.; O'HEARN, P. W. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In: *Proc. SCAM'18*. [S.l.: s.n.], 2018.

HERZIG, K.; NAGAPPAN, N. Empirically detecting false test alarms using association rules. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 39–48. Available at: <http://dl.acm.org/citation.cfm?id=2819009.2819018>.

JMOCKIT. 2021. Https://jmockit.github.io/.

KING, C.; WATERLAND, A. *stress-ng*. 2020. Https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html#description. [Online; accessed April-2020].

KING, T. M.; SANTIAGO, D.; PHILLIPS, J.; CLARKE, P. J. Towards a bayesian network model for predicting flaky automated tests. In: IEEE. *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. [S.l.], 2018. p. 100–107.

KOPYTOV, A. *sysbench*. 2020. Https://github.com/akopytov/sysbench. [Online; accessed Dec-2020].

LAM, W.; GODEFROID, P.; NATH, S.; SANTHIAR, A.; THUMMALAPENTA, S. Root causing flaky tests in a large-scale industrial setting. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019. (ISSTA 2019), p. 101–111. ISBN 9781450362245. Available at: <https://doi.org/10.1145/3293882.3330570>.

LAM, W.; OEI, R.; SHI, A.; MARINOV, D.; XIE, T. idflakies: A framework for detecting and partially classifying flaky tests. In: IEEE. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. [S.l.], 2019. p. 312–322.

LAM, W.; WINTER, S.; WEI, A.; XIE, T.; MARINOV, D.; BELL, J. A large-scale longitudinal study of flaky tests. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 4, n. OOPSLA, Nov. 2020. Available at: <https://doi.org/10.1145/3428270>.

LEINARDI, R. *GtkStressTesting*. 2020. Https://gitlab.com/leinardi/gst. [Online; accessed Dec-2020].

LISTFIELD, J. *Where do our flaky tests come from?* 2017. Https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

LUO, Q.; HARIRI, F.; ELOUSSI, L.; MARINOV, D. An empirical analysis of flaky tests. In: *Proc. FSE'14*. [S.l.: s.n.], 2014.

MANUSKIN, A. *s-tui*. 2020. Https://amanusk.github.io/s-tui/. [Online; accessed Dec-2020].

MEYER, M. Continuous integration and its tools. *IEEE software*, IEEE, v. 31, n. 3, p. 14–16, 2014.

MICCO, J. *Flaky Tests at Google and How We Mitigate Them*. 2016. Https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html.

MICCO, J. *The State of Continuous Integration Testing @Google*. 2017. ICST.

PALMER, J. *Test Flakiness – Methods for identifying and dealing with flaky tests*. 2019. Https://labs.spotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/.

PALOMBA, F.; ZAIDMAN, A.; LUCIA, A. D. Automatic test smell detection using information retrieval techniques. In: IEEE. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2018. p. 311–322.

PEREIRA, L. A. F. *hardinfo*. 2020. Https://github.com/lpereira/hardinfo. [Online; accessed Dec-2020].

PINTO, G.; MIRANDA, B.; DISSANAYAKE, S.; D'AMORIM, M.; TREUDE, C.; BERTOLINO, A. What is the vocabulary of flaky tests? In: *International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2020. To Appear.

RAHMAN, M. T.; RIGBY, P. C. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. (ESEC/FSE 2018), p. 857–862. ISBN 9781450355735. Available at: <https://doi.org/10.1145/3236024.3275529>.

REBERT, A.; CHA, S. K.; AVGERINOS, T.; FOOTE, J.; WARREN, D.; GRIECO, G.; BRUMLEY, D. Optimizing seed selection for fuzzing. In: *USENIX*. [S.l.: s.n.], 2014. p. 861–875.

ROMANO, A.; SONG, Z.; GRANDHI, S.; YANG, W.; WANG, W. An empirical analysis of ui-based flaky tests. In: *Proceedings of the International Conference on Software Engineering*. [S.l.: s.n.], 2021. To Appear.

SEN, K. Effective random testing of concurrent programs. In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2007. (ASE '07), p. 323–332. ISBN 9781595938824. Available at: <https://doi.org/10.1145/1321631.1321679>.

SHI, A.; LAM, W.; OEI, R.; XIE, T.; MARINOV, D. 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*. [S.l.: s.n.], 2019. p. 545–555.

SILVA, D.; TEIXEIRA, L.; D'AMORIM, M. Shake it! detecting flaky tests caused by concurrency with shaker. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020. p. 301–311. Available at: <https://doi.org/10.1109/ICSME46990.2020.00037>.

TEST 001InvalidQRCode. 2021. <https://github.com/0xbb/otp-authenticator/blob/e7a6e4b028e72800cf853bc78362c3e62dee2158/app/src/androidTest/java/net/bierbaumer/otp_authenticator/MainActivityTest.java#L118>.

TEST authenticateWithProvisioning. 2021. <https://github.com/Azure/azure-iot-sdk-java/blob/a9226a5/provisioning/provisioning-device-client/src/test/java/tests/unit/com/microsoft/azure/sdk/iot/provisioning/device/internal/contract/mqtt/ContractAPIMqttTest.java#L428>.

TEST CreateMedicine. 2021. <https://github.com/citiususc/calendula/blob/5c862120e3d607ce2bf828b40785bf361a253710/Calendula/src/androidTest/java/es/usc/citius/servando/calendula/activities/MedicinesActivityCreateTest.java#L83>.

TEST ImagefapAlbums. 2021. <https://github.com/RipMeApp/ripme/blob/9f34861f20a652af50dcec1e4c0ef13e327abc8f/src/test/java/com/rarchives/ripme/tst/ripper/rippers/ImagefapRipperTest.java#L13>.

TEST RetrieveFromFile. 2021. <https://github.com/apache/hbase/blob/d50816fe448971b8e586792f0584aaf601e31780/hbase-server/src/test/java/org/apache/hadoop/hbase/io/hfile/bucket/TestVerifyBucketCacheFile.java#L86>.

TEST vaadinServlet. 2021. <https://github.com/vaadin/flow/blob/df7a5f80b8c77f5d1505ecd6aea7e1adf70141bf/flow-server/src/test/java/com/vaadin/flow/server/DevModeHandlerTest.java#L377>.

THORVE, S.; SRESHTHA, C.; MENG, N. An empirical study of flaky tests in android apps. In: IEEE. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2018. p. 534–538.

TRAN, H. K. V.; ALI, N. B.; BÖRSTLER, J.; UNTERKALMSTEINER, M. Test-case quality–understanding practitioners' perspectives. In: SPRINGER. *International Conference on Product-Focused Software Process Improvement*. [S.l.], 2019. p. 37–52.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. An empirical investigation into the nature of test smells. In: IEEE. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2016. p. 4–15.

VAADIN Flow. 2021. Https://vaadin.com/flow.

VAHABZADEH, A.; FARD, A. M.; MESBAH, A. An empirical study of bugs in test code. In: *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. USA: IEEE Computer Society, 2015. (ICSME '15), p. 101–110. ISBN 9781467375320. Available at: <https://doi.org/10.1109/ICSM.2015.7332456>.

VARGHA, A.; DELANEY, H. D. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, Sage Publications Sage CA: Los Angeles, CA, v. 25, n. 2, p. 101–132, 2000.

VERDECCHIA, R.; CRUCIANI, E.; MIRANDA, B.; BERTOLINO, A. Know you neighbor: Fast static prediction of test flakiness. *IEEE Access*, v. 9, p. 76119–76134, 2021.

WATERLOO, M.; PERSON, S.; ELBAUM, S. Test analysis: Searching for faults in tests. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015. (ASE '15), p. 149–154. ISBN 978-1-5090-0024-1. Available at: <https://doi.org/10.1109/ASE.2015.37>.

WINTERS, T.; MANSHRECK, T.; WRIGHT, H. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020. ISBN 9781492082798. Available at: <https://books.google.com.br/books?id=TyIrywEACAAJ>.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012.

ZHANG, S.; JALALI, D.; WUTTKE, J.; MUSLU, K.; LAM, W.; ERNST, M. D.; NOTKIN, D. Empirically revisiting the test independence assumption. In: *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. [S.l.: s.n.], 2014. p. 385–396.