A Multi-factor Approach for Flaky Test Detection and Automated Root Cause Analysis

Azeem Ahmad
Department of Computer Science
Linköping University
Linköping, Sweden
azeem.ahmad@liu.se

Francisco Gomes de Oliveira Neto
Department of Computer Science
Chalmers and the University of Gothenburg
Gothenburg, Sweden
gomesf@chalmers.se

Zhixiang Shi
Department of Computer Science
Linköping University
Linköping, Sweden
zhish307@student.liu.se

Kristian Sandahl
Department of Computer Science
Linköping University
Linköping, Sweden
kristian.sandahl@liu.se

Ola Leifler
Department of Computer Science
Linköping University
Linköping, Sweden
ola.leifler@liu.se

2021 28th Asia-Pacific Software Engineering Conference (APSEC) | 978-1-6654-3784-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/APSEC53868.2021.00041

***Abstract*—Developers often spend time to determine whether test case failures are real failures or flaky. The flaky tests, also known as non-deterministic tests, switch their outcomes without any modification in the codebase, hence reducing the confidence of developers during maintenance as well as in the quality of a product. Re-running test cases to reveal flakiness is resource-consuming, unreliable and does not reveal the root causes of test flakiness. Our paper evaluates a multi-factor approach to identify flaky test executions implemented in a tool named `MDFlaker`. The four factors are: trace-back coverage, flaky frequency, number of test smells, and test size. Based on the extracted factors, `MDFlaker` uses k-Nearest Neighbor (KNN) to determine whether failed test executions are flaky. We investigate `MDFlaker` in a case study with 2166 test executions from different open-source repositories. We evaluate the effectiveness of our flaky detection tool. We illustrate how the multi-factor approach can be used to reveal root causes for flakiness, and we conduct a qualitative comparison between `MDFlaker` and other tools proposed in literature. Our results show that the combination of different factors can be used to identify flaky tests. Each factor has its own trade-off, e.g., trace-back leads to many true positives, while flaky frequency yields more true negatives. Therefore, specific combinations of factors enable classification for testers with limited information (e.g., not enough test history information).**

***Index Terms*—flaky tests, non-deterministic tests, flaky test detection, automated root-cause analysis, trace-back**

I. Introduction

When test cases fail, developers expect test failures to be connected to the code changes but some of these test failures have nothing to do with the code changes. Developers spend time analyzing changes trying to identify the source of the test failure, only to find out that the cause of the failure is test flakiness. A flaky test is a test that exhibits both passing and failing outcomes when no changes are introduced in a codebase [1], [2]. Flaky tests do not reveal faults in the system under test and the causes of the failure are associated with, e.g., issues with the test execution or the design of the test case, etc. Flaky tests are receiving a great attention from practitioners and researchers due to the challenges that they introduce to software testing and maintenance, such as introducing noise to test execution reports [3]–[8]. The most common approach for flaky test detection is re-running test cases. However, re-running test cases is resource consuming (Google uses 2-16% of its testing budget for rerunning tests [9]) and unreliable [4]. It is a challenge to determine the number of re-runs that can reveal a discrepancy in test outputs [4]. Many efforts have been dedicated to flaky test detection as compared to locating root causes of test flakiness. For example, developers can detect test flakiness by simply rerunning the test multiple times and seeing variations. However, practitioners' main interest is to know what test inputs or configurations lead to test flakiness, thus requiring great attention to a frequently asked question by developers: *"why is this test case flaky?"*.

In this paper, we propose and evaluate a multi-factor approach for flaky test detection named `MDFlaker`. In addition to detection, this technique indicates the reasons why the test cases are flaky. `MDFlaker` consists of four factors: trace-back coverage, test smells, flaky frequency, and test case size. After acquiring information about the aforementioned factors in failed test cases, `MDFlaker` uses a machine-learning algorithm (i.e., k-Nearest Neighbour or KNN) to classify whether failed test executions are flaky or not. Below, we summarise each factor used by `MDFlaker` (details in Section III).

Trace-back coverage: This factor is a modified approach based on the differential coverage proposed by Bell et al. [10]. Trace-back is a lightweight technique that incorporates information from test execution logs and marks a test case as flaky if it did not execute on code changes.

Flaky Frequency: `MDFlaker` maintains a history for each test case where the test's outcome changes without any modification in a codebase. The tests that switch their outcome without any modification in codebase during several occasions in the past has a higher flaky frequency and are more likely to be flaky [11], [12].

Test Smells: Many studies empirically show the relationship between test smells and test flakiness [13]–[15]. In order to leverage from such relationships, `MDFlaker` detects and maintains a database of test cases and the number of test smells present in the test cases.

Test Case Size: `MDFlaker` keeps a database of the lines of code in each test case. Evidence suggests that test case size contributes to test flakiness. For example, test cases with many steps and assertions are often unstable [12]. Ahmad et al. [16] concluded that small tests tend to be more stable, and large flaky tests can be repaired by breaking them down into several small tests.

Individually, the aforementioned factor offers relevant information and, when combined, they offer an aggregated score about test flakiness. We use these scores and information from those factors to prevent future test flakiness as well as to identify and correct current ones. The contributions of this papers are:

- A lightweight technique to find out if failed test cases were executed on code changes, thus determining if test cases are flaky or not.
- An investigation of the efficiency of combining different factors followed by machine-learning to detect test flakiness.
- Examples on how the classification logs can support testers in understanding the root causes of test flakiness.
- A high-level comparison of `MDFlaker` with other flaky test detectors.
- An open-source detector tool named `MDFlaker` [1].

The detailed description, implementation and evaluation of `MDFlaker` (i.e., each factor together with machine-learning algorithm) is presented in Section III, IV, and evaluation V , respectively. Section VI and VII present the discussion and validity threats, respectively.

II. Related Work

Research on flaky tests is mainly divided into causes and classification of flakiness [13], [15], [17], [18], detection of test flakiness [1], [10], [12], [19], and other relevant factors that affect test flakiness [16], [20], [21].

Luo et al. [13] classified flaky tests by analyzing 52 open-source Java projects. They inspected 201 commits of selected projects. Asynchronous wait (45%), concurrency (20%), and test order dependency (12%) were found to be the most common causes of test flakiness. Eck et al. [17] investigated the developers' perceptions on the causes, fixing efforts, significance, and challenges of test flakiness. They conducted their study with Mozilla's' developers to classify 200 real-world flaky tests in terms of the nature of the flakiness, the origin of the flakiness (test or production code), and the fixing efforts. They discovered four new types of causes such as 1) Too restrictive range (i.e., valid output values are outside the assertion range), 2) Test case timeout, 3) Platform dependency and 4) Test suit timeout. The types of causes in those papers are orthogonal to the factors used by `MDFlaker` (e.g., test smells), but our approach focuses on identifying test flakiness instead of classifying the flakiness itself.

[1]https://anonymous.4open.science/r/MDFlakerForPaperUpdated-B003/README.md

King et al. [12] proposed a machine learning approach that uses a Bayesian network model to predict if a test is flaky or not. They view the test flakiness problem as a disease so that they can model its observable impacts as symptoms, and then determine any causal factors for those symptoms. The Bayesian network for test flakiness included symptoms, causal factors and supporting metrics. Lam et al. [1] developed a framework (iDFlakies) to detect and classify certain kinds of flaky tests. The framework was able to detect flaky tests and classify them as either order-dependent (OD) or non-order-dependent (NOD). However, their tool detects flaky tests by reversing the original order of the tests and rerunning which means that the time consumption is much higher than those tools that do not require rerunning tests. Bertolino et al. [19] used a static approach, named FLAST, for flakiness detection using test code similarity. The extensive evaluation on 24 projects taken from different repositories indicated that FLAST can identify flaky tests up to 0.98 Median and 0.92 Mean precision. Different factors that affect test flakiness were investigated by Ahmad et al. [16] using a multiple case study of 5 different companies. They listed 23 unique factors that either increase, decrease or affect the ability to find test flakiness. As mentioned, there are many approaches to identify test flakiness. They mainly differ in which factors are mainly used or which frameworks/languages they support. We briefly compare some of tools with each other as well as with `MDFlaker` in Section VI. Differential coverage is a technique for detecting flakiness proposed by Bell et al. in DeFlaker [10] and adapted to support flaky detection in `MDFlaker`. Differential coverage first analyzes which classes, methods, and statements in the code need to be traced through the modified code, and then monitors whether they are executed by injecting cursors into the code to be traced as classes are loaded into the *JVM* during test execution. Later it is concluded whether the test is a flaky if it is executed on modified code. If the modified code is not executed on the premise that the test passed the last time, the test is considered flaky. The result indicated that Deflaker found 4,846 flaky tests among the 5,075 flaky tests (precision of 95.5%).
`MDFlaker` leverages on the concept of relating test execution with code changes proposed by [10] but does not perform instrumentation on the code execution. Instead, `MDFlaker` mines version control logs to match test executions and code changes. In summary, the main distinction is that differential coverage entails code instrumentation, whereas our approach is offline. Mining code changes removes the need to inject a cursor to monitor the execution of the code which increase the execution time. For instance, Bell et al. claimed that their technique increases the running time of a single test by 4.6% on average. Moreover, our approach samples only the failed tests but differential coverage will monitor the execution of all tests which causes additional and unnecessary

overhead for flaky test detection. Furthermore, implementing differential coverage in other programming languages requires re-customization of the classes, methods, and declarations that need to be tracked for detection, as well as the function of monitoring code execution, also needs to be re-developed making it more complicated to migrate the differential coverage technique to projects with other or more than one programming language. In contrast, our approach would need specific parsers to map commit and trace-back logs in different programming languages.

III. Multi-factor Flaky Detection

We present the details of each factor along with an overview of how they compose the architecture used for *MDFlaker*.

A. Trace-back Coverage

The core idea of the trace-back coverage is to locate the changes in the code base and identify if the failed test cases execute those changes. Failed test cases that do not traverse code changes are likely to be flaky [10]. Nonetheless, this assumption is also connected to the granularity of the changes (e.g., file-level, methods, line of code). When a build fails, `MDFlaker` searches through the logs to identify the files or methods that triggered the failures in the code.

The information extracted from logs is referred to as trace-back. These trace-backs indicate which methods and statements caused the failure, and usually include three important information used by testers in debugging: file path, method name, and statement line. Listing 1 shows an example of a trace-back from a failure in a Python project [2]. The failure of the test method (*test_load_birth_names*) calls another test method (*load_birth_names*) in line 41 of the test file (*load_examples_test.py*). Furthermore, the failure of test method (*load_birth_names*) is due to the call to another test method (*load_data*) in line 84 of the test file (*birth_names.py*). Parsing through trace-back information, we can get the error chain revealing whether the failed test methods were executed on the production code changes. This information about whether the test code reached changes in production code helps `MDFlaker` decide whether failed test cases are flaky.

```
ERROR: test_load_birth_names (tests.load_examples_test.SupersetDataFrameTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):

File "/home/travis/build/apache/incubator-superset/tests/load_examples_test.py",
    line 41, in test_load_birth_names
self.example.load.birth_names()

File "/home/travis/build/apache/incubator-superset/superset/examples/birth_names.
    py",  line 84, in load_birth_names load_data(tbl_rame, database)

File "/home/travis/build/apache/incubator-superset/superset/examples/birth_names.
    py",  line 57,  in load_data pdf = pd,read_json(get_example_date("birth_names
    .json.gz"))

File "/home/travis/build/apache/incubator-superset/superset/examples/helpers.py",
     line 223, in get_example_data content = request.urlopen(f"{BASE_URL}{
    filepath)?raw=true).read()

File "/opt/python/3.6/lib/python3.6/urllib/request.py", line 73, in urlopen
    return opener.open(url, data, timeout)
.........................

```

Listing 1: An example of a trace-back from a failure in a Python project

[2]https://github.com/apache/incubator-superset

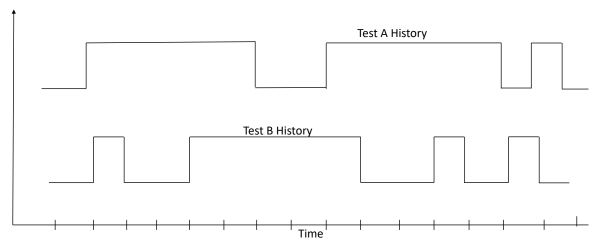


Fig. 1: Build history of test A and B. Each version indicates whether a test case passed (0) or failed (1).

One of the limitations of trace-back, as presented in Listing 1, is that it reveals only method calls. For example, if a commit only changes a specific line of code within a test method (e.g., parameters or values in a variable), then the trace-back will not reveal these changes which might lead to a misleading classification of a flaky test since the actual changes may not be visbile at the method call level. Therefore, we complement the trace-back by mining additional information from the commit logs that include detailed code changes in a version control repository. Whenever there is *not* an intersection between (i) the set of methods from a failure trace back and (ii) the set of parsed code changes, we tag the corresponding test case as flaky. In other words, the intersection indicates that the trace-back indeed covers the changes (i.e., not flaky), whereas no intersection indicates that the failure trace-back does not cover the changes (i.e., flaky tests)

B. Flaky Frequency

The flaky frequency factor conveys the relationship between the failed build histories and flaky tests. We refer to build history as the results of all test sessions for the previous builds of the software system (e.g., as part of a Continuous Delivery pipeline). Since each commit represents a certain degree of modifications in the codebase, it is risky to directly link flaky tests and the number of test failures or state transitions (i.e., builds where the test outcome changed from failure to pass, or vice versa). Figure 1 shows different build histories of the two tests (TA and TB). Test failures (cells that contain 1 in Figure 1) that extend over multiple builds indicate that the builds have failed continuously in that period (e.g., a fault that could not be fixed and kept being triggered), whereas an isolated failure indicates that the test failed once at a certain point in time.

We are interested in those occurrences in which a test history was passing and suddenly changes to failure. For instance, TA has switched to failure (1). Similarly, TB switched to failure a total of four times: one failure 3 times periods and 5 failures in one time period. Several reasons (e.g., external dependency issues, improper test setup, etc,) are associated with builds to be continuously failed for a longer period. We do not intend to list these reasons as it is out of the scope of this paper, rather we focus on the changes in order to help `MDFlaker` to predict flaky tests.

Different studies [11], [12] have investigated the relationship between changing test outcomes and test flakiness. Liviu [11] stated that a test with a build history similar to *TB* is more

TABLE I: Test Smells used by *MDFlaker* as proposed in [14].

Test Smells	Description
Async wait	Includes tests that make an asynchronous call and do not wait properly for the result of the call to become available before starting to use it.
Precision	Flakiness comes from an assertion that was too hard to achieve or using a too narrow range of valid values (e.g., floating point errors).
Randomness	This implies that the test was flaky due to improper use of a random number generator (e.g., undefined seeds).
Network	Tests whose execution depends on the network can be flaky because the network is a resource that is hard to control.
Unordered Collections	Tests that assume that the elements are returned in a particular order when iterating over unordered collections.
IO	This flakiness emanates from problems with file descriptors or resources that do not exist.
Training	The common actions that can cause this kind of flakiness are training with little data or few iterations and using a bad network dimension that is not suited for the task.
Time	The tests are flaky due to time-related problems.

likely to be a flaky test. King et al. [12] used a flip ratio to indicate the stability of the test. The flip rate is the number of state transitions of a test in a period divided by the total number of test runs. We used another way to express build history which we call flaky frequency. We consider a test to fail multiple times in a short period as one state transition. The flaky frequency is equal to the number of state transitions (from pass to fail) in a period divided by the total number of failures (see equation 1). The denominator is set to a number of failures + 2 to prevent the flaky frequency of the new failed test (the recently added test in the test suite) from being equal to 1. Considering the equation 1, the flaky frequency of *TA* is about 0.23 (3/13) and the flaky frequency of *TB* is 0.4 (4/10).

$$Flaky\,Frequency = \frac{Number\,of\,state\,transitions}{number\,of\,failures + 2} \qquad (1)$$

C. Test Smells

Test smells are bad design or improper implementation methods in the process of test writing. Many studies have identified flakiness-inducing test smells and concluded that different test smells correspond to different types of flakiness [13], [15], [16], [22]. Thus, we consider it an important factor for flaky test detection. Sjöborn [14] investigated open-source *Python* projects and cataloged 11 root causes of test flakiness including eight categories related to test smells and three categories related to infrastructure, concurrency and test order dependency. Table I presents the eight categories from [14] used in *MDFlaker*. The list of test smells listed in literature are many and too fine-grained such that, the direct connection to flaky test categories is not clear. Therefore, we aggregated some of those test smells from literature into categories that directly relate to flaky factors

D. Test Case Size

The test size is the number of physical lines of code in the test script. King et al. [12] assumed that large test scripts with many steps and assertions are often unstable. Ahmad et al. [16]

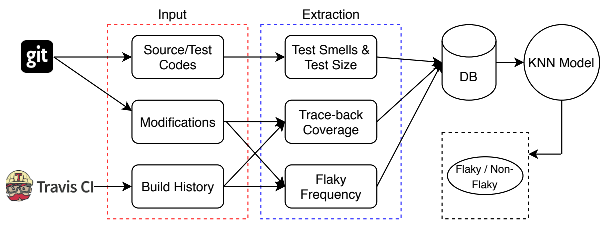


Fig. 2: A high level architecture of *MDFlaker*.

showed that small tests tend to be more stable and many large flaky tests can be repaired by breaking them down into several smaller tests. Moreover, when a test covers multiple scenarios, the probability of it becoming flaky is likely increase. Thus, we consider it a promising factor to identify test flakiness and included it in `MDFlaker`.

IV. `MDFlaker` - Architecture and Implementation

The implementation of `MDFlaker`[3] is written in Python version 3.6. We used different libraries such as `ast` for converting Python codes into abstract syntax tree, `re` for logs/build analysis, `PyMySQL`[4] for database operations, `ski-learn`[5] for machine learning algorithm, and `matplotlib`[6] for plotting and visualisation. Figure 2 presents *MDFlaker*'s architecture.

Measures pertaining test size and test smells are extracted from source/test code in GitHub. The trace-back coverage is calculated through build history from Travis and modifications from GitHub. Lastly, flaky frequency is calculated using build history and modifications from GitHub. All data is stored in a database. We used the KNN model to predict if failed test executions are flaky or not. The following sections describe how we calculate/implement each factor in detail.

A. Trace-back Coverage and Flaky Frequency

The important sources to calculating trace-back coverage and flaky frequency are: build history from Travis CI and modifications made in the codebase (i.e., commits) from GitHub as shows in Figure 2. The tools mines `json` files from Travis CI that contain information about the different builds. Each build contains at least one job. Each job defines a different environment and test conditions to building and testing of the project under test. For example, each job can use a different version of Python to build the project. A job also maps to a log that includes detailed information of the build such as the process of setting up the environment, the dependencies of the installation, the test process, executed classes and method, and results, etc. Therefore, each build and job correspond to a unique number.

We obtain the failed build logs from Travis CI, then we mine failed tests from the logs. Particularly, we extract the names

[3]https://anonymous.4open.science/r/MDFlakerForPaperUpdated-B003/README.md
[4]https://pypi.org/project/PyMySQL/
[5]https://scikit-learn.org/stable/
[6]https://matplotlib.org/

of methods, classes, files, and file paths. Then, a script parses the trace-back information (an example was shown in Figure 1) corresponding to each failed test. Each trace-back includes the test or tested method name, the line number where the statement (i.e., where the failure occurs) is located, and the file path to which the method belongs. Finally, the data is stored in `json` format for subsequent processing.

Once we identify the failed tests and trace-back information, we needed code or test changes from GitHub. Using unique *SHA* values from each failed build, `MDFlaker` extracts code changes (as detailed in Section III-A). If the changes from GitHub and trace-back overlap, it represents that failed test cases exercised the changes. If there is no overlap, it means that the trace-back did not cover the changes, hence indicating a possible flaky test execution. `MDFlaker` uses similar sources (e.g., Travis CI and GitHub) to calculate flaky frequency. Since we have failed builds logs and code changes, we search for test outcome changes in all previous builds. The flaky frequency value of each test case is maintained in a local database and updated when new information is available.

B. Test Smells and Test Case Size

We wrote a parser to extract flakiness-inducing test smells in *Python* test cases as discussed in Section III-C. Most of the flakiness-inducing test smells were obtained through searching for single keywords in the test code (e.g., *sleep*, `wait`, `float`), whereas others require searching of multiple keywords such as the smell random with or without seed, use open with or without close, among others. For efficiency and correct test code parsing, we use the Abstract Syntax Tree library (*ast*) provided by Python itself. The *ast* is an intermediate product from source code to byte-code which can help us analyze the source code structure from the perspective of the syntax tree. For instance, to match `time.sleep`, we did not match the text "time.sleep", rather we search for the actual function *time.sleep*.

In turn, the test case size refers to non-commented lines of test case code. We wrote a simple script that counts the number of non-commented lines in all test case codes. We understand the risks of counting physical lines of codes instead of, e.g., logical lines of code. However, physical lines of codes are simpler to calculate and test code often calls on the System Under Test, such that complex logical constructs (e.g., several nested function calls in the same line) are less likely to occur when compared with production code. We aim to expand the test size factor to include other size measures (e.g., perhaps chosen by a tester more familiar with the test code) in future studies.

C. K-nearest Neighbor (KNN)

Using the four different factors above, we use K-nearest neighbor (KNN) to classify whether a test case is flaky. KNN is a non-parametric statistical machine learning algorithm for classification and regression that relies on labeled input data to learn a target function that produces an appropriate output when given new unlabeled data [23]. In principle, KNN puts all correctly classified data for training into a feature space and classifies the test data by measuring the distance (i.e., *Manhattan Distance* in this study) between different features values. Given a test instance i, find the k nearest neighbors and their labels and then predict that label of i is the majority label of the k nearest neighbors. Usually, k is an integer not greater than 20.

We choose KNN because (i) new data can be directly added to the data set without retraining the model, and (ii) KNN has no pre-assumptions about the data. Moreover, the model tends to show high accuracy and is not as sensitive to outliers as other approaches. Nonetheless, since the classification of a test case can vary over time (e.g., from flaky to non-flaky after refactoring), we decided to train our KNN algorithm on test executions, regardless of the corresponding test case, i.e., the input for the KNN is based on the four factors (trace-back, flaky frequency, test smells and test size). Using test executions also reduces the risk of overfitting in our approach, i.e., we reduce the risk that the KNN algorithm becomes biased towards a specific set of test cases.

In our study, we investigate the impact that each factor and corresponding combinations have in classifying test flakiness. Therefore, we evaluate the classification capabilities of the KNN with all combinations of factors. We explain the steps for training and testing of the KNN model when detailing our evaluation (Section V).

V. Evaluation and Results

We evaluate our approach and tool in a case study with open-source projects. Our goal is to assess which factors are more effective to identify flaky test executions, whether those factors help in identifying root causes for test flakiness and, lastly, how does `MDFlaker` compare with other existing tools that classify test flakiness. The re-analysis package[7] includes the evaluation data and scripts with calculations of precision, recall and accuracy used in our paper. We investigate the following research questions:

RQ1: To what extent do the chosen factors reveal test flakiness? Are there more effective combinations of factors, or are some factors better in isolation?
RQ2: What type of information can *MDFlaker* reveal to developers to help understand the root causes of test flakiness?

A. Data collection and Analysis

Figure 3 presents the workflow of data collection which includes creation of our ground truth set (Step 0), training of the KNN model (Step 1) and evaluation of flaky detection performed by `MDFlaker` (Step 2). We selected three Github projects (i.e., spaCy [8], incubator-superset[9], and django-rest-framework[10]) to run our case study based on different factors.

[7]https://figshare.com/s/3ddb271e2ae71e907c6b
[8]https://github.com/explosion/spaCy
[9]https://github.com/hyperconnect/incubator-superset
[10]https://github.com/encode/django-rest-framework

TABLE II: An overview of the number of artefacts used in our evaluation of `MDFlaker`. The data include the number of versions mined from Github, the total number of extracted failures and the number of failures and flaky tests used in the evaluation subset (i.e., 25% of the dataset).

Project Name	**Versions**	**Extracted failures**	**Subset Failures**	**Subset Flaky**
django-rest-framework	66	583	153	79
incubator-superset	84	660	171	103
spaCy	62	923	239	146
Total	212	2166	563	328

spaCy is a library for advanced Natural Language Processing used by 23k users with 5k contributors, and a star rating of 29k. django-rest-framework is a powerful and flexible toolkit for building Web APIs with 215k users, 1.1K contributors, and a star rating of 20.8k. incubator-superset is a modern, enterprise-ready business intelligence web application with 24 users, 580 contributors, and a star rating of 38.2k. We chose those three projects because they are reasonably large and belong to a variety of business domains.

We selected a total of 212 versions that had at least one failing tests from the three projects, leading to a set of 2166 test failures. In order to compare our classification, we need to know which test executions from the selected versions showed flaky behaviour, therefore, we checked out and executed each build 30 times in isolation (tests were ran in random orders in order to cover some of the flaky categories defined in literature). Whenever a test would change its outcome during those 30 runs, we would classify the test execution as flaky. From the test set, failures were obtained from these experiments, in which 1372 were flaky tests and 794 were non-flaky tests. That is a very high proportion of flaky test executions, however, recall that our entire set is composed only of failing tests. In other words, our evaluation does not look at any test case that passed, which comprises the majority of the original test repositories in Github. The data-set is shown in Table II. Throughout those 30 executions, the test cases ran in different order (including the original order). Therefore, our ground truth set of flaky tests is limited to flaky factors associated to test executions (dependant order, or infrastructure). Unfortunately, inspecting each test case to detect complementary flaky test categories (e.g., test smells) is infeasible for the amount of projects and tests.

In parallel to building the ground truth, we randomly divided the 2166 tests into training data sets ($75\% \approx 1603$ failures) and test data sets ($25\% \approx 563$ failures). To increase accuracy, choosing the right value of K in KNN algorithm is very important. When K=1, the test examples were given the same label as the closest example (e.g., in the training set) whereas, when K=3, the labels of the three closest classes are checked and the most common (i.e., occurring at least twice) label is assigned. Since, we have four factors, we set K to $factors+1$ (i.e., K=5).

Since we aim to investigate the impact of all combinations of factors (RQ1), we use a full factorial design on all four factors (trace-back, flaky frequency, test smells, test size), in which the corresponding levels are either are enabled (1) or disabled (0) during classification. Since there is a high proportion of flaky test executions in our data set, we also use a random classification as a baseline comparison[11]. For each project, we measure the precision, recall and accuracy as dependent variables to capture the effectiveness of the flaky detection. The measures verify whether `MDFlaker`'s classification matches the actual classification in our ground truth (true positive or true negative—TP or TN) versus the flaky tests that were missed (false negative—FN) and the executions executions that were indeed non-flaky (false positives—FP).

We chose those dependent variables as they have been used in literature, hence enabling the comparison of our results [10], [19]. Moreover, there is often a cost associated with false positives and false negatives. Non-flaky test wrongly classified as flaky would rise to a somewhat insignificant problem because an experienced user can bypass the warning by looking at test case code. Whereas when a flaky test is wrongly classified as a non-flaky test, this is obnoxious, because it indicates the test suite still has test cases whose outcome cannot be trusted.

In turn, RQ2 involve qualitative assessment of the flaky classification. For RQ2, `MDFlaker` we illustrate how the logs generated by `MDFlaker` can support root cause analysis by relating specific properties (e.g., specific types of test smells) and the flaky test classifications. Next, we detail the results for each research question.

B. RQ1—Analysis of Multi-factor Flaky Test Detection

Table III shows insights about our dependent variables related to individual and combined factors in revealing test flakiness. By comparing the results we see that the different factors are more effective in identifying flaky tests than a random approach, despite the high proportion of flaky tests in our evaluation subset (circa 55%). Overall, the results between factors are somewhat similar, differing in 0.1–0.2 which, for our dataset, indicates *roughly* a difference between 5-20 test executions classified correctly by `MDFlaker` (i.e., TP or TN). In turn, a 0.01 difference in any of the dependent variables is negligible.

By comparing accuracy and precision, we notice that all combinations involving trace-back has higher precision and accuracy, even when the factor is considered in isolation. Particularly, frequency seems to be a good combination with trace-back. This is not surprising as both factors focus on the build and change history related to the execution. However, note from Table III that frequency is better at finding true negatives (i.e., non-flaky tests), as opposed to true positives. Moreover, test smells is comparable to random detection when compared for accuracy and precision with the lowest result (0.55), due to the large amount of false positives detected in all cases (despite finding many true positives). This is not surprising as literature has shown that test smells should not be used as the *only* indicator to reveal test flakiness [16], [21].

[11]This would correspond to the trial of all factors disabled.

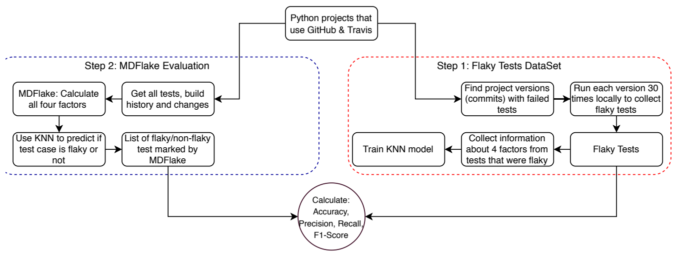

Python projects that use GitHub & Travis
Step 2: MDFlake Evaluation
MDFlake: Calculate all four factors
Get all tests, build history and changes
Use KNN to predict if test case is flaky or not
List of flaky/non-flaky test marked by MDFlake
Step 1: Flaky Tests DataSet
Find project versions (commits) with failed tests
Run each version 30 times locally to collect flaky tests
Train KNN model
Collect information about 4 factors from tests that were flaky
Flaky Tests
Calculate: Accuracy, Precision, Recall, F1-Score

Fig. 3: A detailed workflow about data collection for *MDFlaker*'s evaluation.

TABLE III: Overview of accuracy, precision and recall for each combination of factor and a random classification as a comparison baseline. As a reference, we also included the number of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN) based on the `MDFlaker`'s classification.

Row	Factor	FN	FP	TN	TP	Accuracy	Precision	Recall
1	frequency	81	75	160	247	0.72	0.76	0.75
2	size	38	163	72	290	0.64	0.64	0.88
3	smells	29	222	13	299	0.55	0.57	0.91
4	trace-back	19	73	162	309	0.83	0.80	0.94
5	size, frequency	77	70	165	251	0.73	0.78	0.76
6	smells, size	69	121	114	259	0.66	0.68	0.79
7	smells, frequency	77	72	163	251	0.73	0.77	0.76
8	trace-back, size	27	76	159	301	0.81	0.79	0.91
9	trace-back, smells	18	76	159	310	0.83	0.80	0.94
10	trace-back, frequency	36	60	175	292	0.82	0.83	0.89
11	size, smells, frequency	77	63	172	251	0.75	0.79	0.76
12	trace-back, size, frequency	31	56	179	297	0.84	0.84	0.90
13	trace-back, size, smells	31	64	171	297	0.83	0.82	0.90
14	trace-back, smells, frequency	58	43	192	270	0.82	0.86	0.82
15	trace-back, smells, frequency, size	33	48	187	295	0.85	0.86	0.89
16	random	166	124	111	162	0.48	0.56	0.49

The impact on the classification between flaky and non-flaky is better seen when comparing the results above with the recall values. In contrast to precision and accuracy, recall values tend to be higher than 0.9, particularly when using test smells or trace-back as factors. There are two reasons behind the difference between the results in recall and precision/accuracy. First, spaCy has a high proportion of flaky tests which is increasing the overall precision of our analysis and, secondly, recall does not count false positives and is more sensitive to false negatives which, in turn, tend to be more harmful to the tester. Lastly, using size as a factor leads to false positives and false negatives, hence hindering all of the metrics.

In order to understand the effect of the project, Table IV shows the results per project. Some patterns reveals more insight about the different factors. For instance, spaCy has very high recall due to the high percentage of flaky tests in the project, such that the majority of high recall seen in the aggregated data (Table III) is a consequence of the spaCy recall. Nonetheless, the findings on test smells having overall high recall remains, since that factors is ranked higher in recall for the other two projects analysed.

Considering the django and incubator projects, frequency has lower accuracy and precision when used in isolation but much higher results when combined with trace-back, once again indicating that both history-based approaches are complementary in identifying TP and TN. Based on our results, we recommend that test smells should be investigated together with history-based information (frequency or trace-back). Similar recommendations have been mentioned in literature [16], [21] as the evidence suggests that the efficiency of test smells in revealing test flakiness depends on the project under investigation [20], [21]. Our study indicates that size is not a reliable predictor, which is contrasting to practitioners view where test size is a factor to identify flakiness [16].

RQ1: Results show that combining all factors often yields high precision, accuracy and recall (0.78–0.98). However, since different combinations yield varied results, we argue that the main drivers to choose the combination of factors should be the availability of artefacts (e.g., practitioners should avoid using frequency if history information is not available or scarce). Trace-back and frequency complement each other in finding, respectively, true positives and true negatives. Test smells tend to yield more false positives but less false negatives than other factors, hence test smells can be combined with the history-based approaches due to its high recall. Size is not reliable as a predictor as it yields many false positives and false negatives.

TABLE IV: Results for accuracy, precision and recall for each project and all combinations of factors, i.e., trace-back (tb), frequency (freq), size and test smells. The table also includes, for each project, the total number of test executions classified along with the number of flaky test executions.

django-rest - Total: 153 executions, Flaky executions: 79

Row	Factor	Accuracy	Precision	Recall
1	freq	0.58	0.69	0.36
2	size	0.64	0.60	0.86
3	size, freq	0.62	0.69	0.48
4	size, smells, freq	0.64	0.72	0.49
5	smells	0.50	0.51	0.93
6	smells, freq	0.63	0.73	0.45
7	smells, size	0.62	0.61	0.73
8	tb	0.90	0.89	0.93
9	tb, freq	0.88	0.91	0.84
10	tb, size	0.88	0.90	0.86
11	tb, size, freq	0.87	0.91	0.83
12	tb, size, smells	0.88	0.90	0.86
13	tb, smells	0.90	0.88	0.93
14	tb, smells, freq	0.80	0.91	0.68
15	tb, smells, freq, size	0.84	0.91	0.78
16	random	0.53	0.55	0.51

incubator-superset - Total: 171 executions, Flaky executions: 103

Row	Factor	Accuracy	Precision	Recall
17	freq	0.67	0.73	0.71
18	size	0.61	0.64	0.80
19	size, freq	0.67	0.77	0.66
20	size, smells, freq	0.67	0.77	0.65
21	smells	0.57	0.60	0.86
22	smells, freq	0.67	0.75	0.69
23	smells, size	0.67	0.71	0.76
24	tb	0.89	0.93	0.88
25	tb, freq	0.84	0.93	0.80
26	tb, size	0.81	0.83	0.86
27	tb, size, freq	0.86	0.92	0.84
28	tb, size, smells	0.87	0.91	0.86
29	tb, smells	0.88	0.92	0.89
30	tb, smells, freq	0.79	0.92	0.71
31	tb, smells, freq, size	0.88	0.93	0.86
32	random	0.49	0.58	0.56

spaCy - Total: 239 executions, Flaky executions: 146

Row	Factor	Accuracy	Precision	Recall
33	freq	0.84	0.80	0.98
34	size	0.66	0.65	0.95
35	size, freq	0.85	0.81	0.99
36	size, smells, freq	0.87	0.83	0.99
37	smells	0.56	0.59	0.93
38	smells, freq	0.84	0.80	0.97
39	smells, size	0.68	0.70	0.83
40	tb	0.74	0.71	0.98
41	tb, freq	0.78	0.74	0.97
42	tb, size	0.77	0.73	0.98
43	tb, size, freq	0.81	0.77	0.98
44	tb, size, smells	0.76	0.74	0.95
45	tb, smells	0.74	0.71	0.98
46	tb, smells, freq	0.84	0.81	0.97
47	tb, smells, freq, size	0.84	0.80	0.98
48	random	0.44	0.55	0.43

C. Revealing Root Causes of Test Flakiness: RQ2

Re-running test cases can indicate which are flaky, but it does not provide sufficient or no evidence about the root causes of test flakiness. As part of the reporting, `MDFlaker` provides different types of information (e.g., CSV files or graphs) about either each test case or a complete test suite that can help them resolve test flakiness or at least provide an indication about where to start looking. Figure 4 presents an image generated by `MDFlaker`. This image reveals the proportion of test smells in a project under investigation.

MDFlaker also creates tables with detailed information about the test executions classified as flaky, such as test case name, the number of test smells, types of test smells, line number, and file name as presented in Figure 5. By visualising additional information connected to failures, practitioners are more equipped to debug [24]. In fact, many researchers have concluded that there is a strong relationship between test smells and test flakiness [13], [15], which we also measured when investigating RQ1. Literature proposes many solutions to fix test cases that were flaky due to test smells. Developers may use this information about test smells present in their test cases and plans to either eliminate test smells or fix anti-patterns in order to eliminate the effect of test smells. For instance, Ahmad et al. [21] suggests adding seeds to the test code to cancel the effect non-determinism in test cases.

Human factors also play a role in fixing test flakiness as developers and testers blame each other for the causes of test flakiness [16]. For example, testers claim that the flakiness is due to changes in production code whereas developers claim that test case code is flaky. Therefore, one suggestion is to use the detailed information about flakiness to strengthen the responsibility of team members and stimulate agency in addressing the flakiness. An example is using the information such as whether failed test case was executed on the latest code changes or not (e.g., trace-back), the number of tests smells in the test case, the flaky frequency can help in resolving such conflicts. We demoed the tool and technique (i.e., *MDFlaker*) to three developers from two different companies to inquire the importance and usefulness of the presented/reported information. All developers suggested that these types of indications are useful to debug test flakiness.

RQ2: Testers spend from several minutes to days to identify the root causes of test flakiness manually [16]. The aggregated information from *MDFlaker* related to test executions helps tester to investigate the root causes of flakiness manually by providing useful indications, such as a subset of test cases containing test smells and the categories/distribution of those smells.

VI. Discussion and Implications

Automated Root Cause Analysis of Flaky Tests: As compared to flaky test detection, literature about automated analysis of root cause of test flakiness is scarce. The challenge

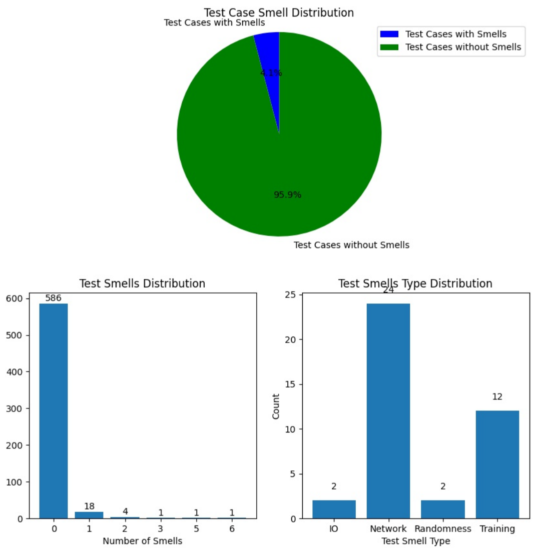
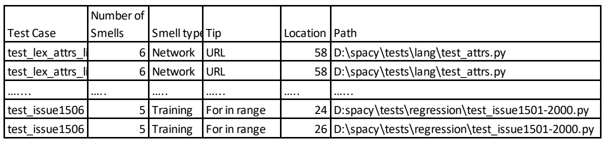


Fig. 4: Images generated by *MDFlaker* to present different types of information about test smells in test cases.



Test Case	Number of Smells	Smell type	Tip	Location	Path
test_lex_attrs_li	6	Network	URL	58	D:\spacy\tests\lang\test_attrs.py
test_lex_attrs_li	6	Network	URL	58	D:\spacy\tests\lang\test_attrs.py
…….	…..	…..	…...	…..	…...
test_issue1506	5	Training	For in range	24	D:spacy\tests\regression\test_issue1501-2000.py
test_issue1506	5	Training	For in range	26	D:\spacy\tests\regression\test_issue1501-2000.py

Fig. 5: A Snapshot from CSV File: Detailed Description of Test Smells, their Type, Test Case Name, Line Number and File Name from the Project Under Investigation

in the flaky test is not detection but finding the root cause of test flakiness. Re-running or other techniques may help in labeling flaky tests but researchers and practitioners should dedicate efforts in findings automated techniques in determining the root cause of test flakiness. Such techniques can answer the question: *"Why is this test case flaky?"*. *MDFlaker* is an alternative to manual activities (i.e., looking at code coverage, number of test smells, and the history of test cases) and provides sufficient information to testers to determine whether the test case failure is a flaky or a real failure. *MDFlaker* produces different types of graphs and CSV files, related to code coverage, the number of test smells, types of test smells, test case size, and test case history for being flaky, for further analysis. That information also helps to avoid the blame-game [16] where flakiness in production or test code introduces friction between testers and developers.

Does Machine Learning Help? Precision is a combination of the classifier and the dataset. It is not wise to ask how precise a classifier is in isolation but to ask how precise a classifier is for a given data set. The reported recall in our study (i.e., 91% for multi-factor and 94.0% for trace-back) means that 9% or 6% of what was flagged as flaky was not flaky. The question developers should ask: *How much did the effort require to address these false positives?*. The multi-factor approach provides 48 instances of false positives where flaky test case was marked incorrectly as non-flaky tests. However, *MDFlaker* would still provide factors information (i.e., test smells, test history, etc.) to testers that can still be helpful to either ignore or give attention to the flags. Similarly, the multi-factor approach provides 33 instances of false-negative where non-flaky tests were marked incorrectly as flaky tests but testers can still utilize the root cause analysis to ignore the warnings. *MDFlaker* can only provide indications and final authority to flag the test case as flaky or non-flaky rests with testers/developers. Machine learning techniques should explicitly discuss the impact of their precision and recall in the actual risks in the classification translating what impact does more or less false positives/negatives have on the tester's daily activities. The discussion about the importance of precision over recall should be raised among team members and should be relevant to the business needs.

Flakiness in a Production Code We speculate that the reason for lower accuracy, precision, and recall values of some of the factors (i.e., individuals or combined) might be due to lack of information available about production code. Similar information (i.e., four factors) about production code is as important as for test case code. Combining factor information from both the source code and test case code might increase the accuracy, precision, and recall. The burden of flakiness should be taken away from test cases and practitioners and researchers should pay attention to production code flakiness. Many researchers [13], [15], [21], [25] investigated test case code to identify test smells inducing test flakiness whereas the investigation about flakiness in production code is scarce.

Specific Flakiness Category - Specific Solution: Many studies [13], [15], [21], [25] have been conducted to list the categories of test flakiness such as `Async Want, IO, Concurrency, Randomness` but the literature is limited when providing suggestions on how to address specific categories. Only few studies such as [4], [18] provides solution to address specific categories such as concurrency. We believe that the focus of test flakiness should move towards providing stare-of-art techniques to address specific flakiness categories. The effort in detecting flaky tests creates trade-off between different tool support, and its unlikely that a single tool will be able to address all the needs that a project has. Therefore, its important to broaden to more inclusive approaches that can be implemented by different technologies to allow exploration of more categories of test flakiness. Not all industries face similar

TABLE V: Comparison of MDFlaker with other tools

Techniques	Lang.	Train.	Rerun TCs	Time	Hist. Info.	Acc.
Deflaker [10]	Java	No	Yes	4.6%	Yes	95.5%
FLAST [19]	Python	Yes	No	NA	1s	92%
Multi-factor	Python	Yes	No	3.4s	Yes	90%[1], 86%[2]
Trace-back	All	Yes	No	3.2s	Yes	90%[1], 84%[2]
Bayesian Network [12]	NA	No	No	NA	No	65.7%

[1] High accuracy from individuals projects
[2] High accuracy from all projects combined

root causes of test flakiness. For example, a survey with 5 different companies conducted by Ahmad et al. [16] revealed no root causes related to `Floating Point Operations, Resource Leak, and Unordered Collection`.

Implications of our findings to practitioners and researchers: Practitioners can use our techniques to detect flaky tests as well for automated root cause analysis. The CSV files and graphs generated by this tool provide relevant information to support practitioners in reducing test flakiness. The `MDFlaker` can be used in many ways: (1) predict which test cases are flaky/non-flaky in a specific failed build, (2) Given the build history and modifications, it can parse all test cases in the test suite repository and predict how many of test cases are flaky/non-flaky in all failed builds, (3) Generating CSV files for available test smells in all test cases, etc. When it comes to supporting future research, our study is a stepping stone towards the alignment of efforts to investigate automated root cause analysis of test flakiness. In addition, to know which test cases are flaky, developers want to know: *why* the tests are flaky.

Comparing *MDFlaker* with Other Tools: `MDFlaker` is inclusive of trace-back technique (i.e., unique contribution of this paper) but we intend to discuss both the `MDFlaker` and trace-back separately, in case developers plan to use trace-back only. Table V presents the comparison of *MDFlaker* with other tools/techniques in flaky test detection with respect to (1) project language, (2) required data set for training or evaluation, (3) test cases require re-running, (4) time to run the technique, (5) need of historical records about test case executions and (6) accuracy. We chose those factors because (i) they have been discussed in previous study [19] as advantages or disadvantages of proposed tools and (ii) they showcase the various properties that differ from various projects such as technical constrains (e.g., programming language) and availability of artefacts (e.g., history, data to train models). Deflaker is a tool that provides high accuracy 0.95 accuracy when compared to the other tools. Our multi-factor approach provides an average of 0.85 accuracy when analysed for all projects, mainly due to the trace-back approach which is an adaption of the classification done by Deflaker. However, there are three advantages of using `MDFlaker`. First, proposed techniques do not add additional burden (i.e., test case instrumentation in Deflaker) on the test case. MDFlaker only needs to collect the historical information after the test cases are executed. Second, the overall time cost of our proposed approach (i.e., trace-back =3.2s and multifactor 3.4s) is smaller than Deflaker because Deflaker has a an overhead of 4.5% on average [10]. The aforementioned time for *MDFlaker* reflects the time taken to extract information about all four factors from each project on average. Third, MDFlaker is very simple to implement as it only requires historical information and applicable to any project written in any language. *MDFlaker* requires only test case parser to extract test case information (i.e., test smells, and test size) to support any language whereas DeFlaker requires instrumentation of source code. Besides, *MDFlaker* does not require the test to re-run thus saving testing time and budget as compared to Deflaker. Flast, on the other hand, provides better accuracy and does not require re-running test cases but it does not provide automated root cause analysis of flaky tests. In addition, Flast is not recommended to use as an alternate to dynamic approaches (i.e., re-running test case or tools that include code coverage) [19]. The difference in accuracy between *MDFlaker* and other tools is not significant thus promising a good addition to tooling within companies.

VII. Validity Threats

Construction Validity: It refers to the choice of factors used in this study. We adopted all factors that have been proven to have an association with flakiness in a variety of previous studies. Also, our choice of projects is limited but the projects were similar in their setup and technologies but different sizes. Future work aims to replicate the study with many more projects

Conclusion Validity We did not use statistical analysis, since our data was measured in terms of accuracy, precision and recall which are percentages and not counts which is the type of measure used by tests like chi-square. Moreover, we did not have a large sample to ensure statistically significant differences. We mitigate those risks, by doing a sensitivity analysis in which we compare the amounts of false positives and negatives in relation to the values of our dependant variables.

External and Internal: Three projects were used in this study for evaluation. These projects may not be representative and affect the generality of these results. The reason for the limited number of projects is because the experiment requires re-running each project 30 times locally to find flaky tests. This was a time-consuming process.

Reproducibility We have released our tool with Apache 2.0 license to enable researchers to reproduce the results. We also provide the processed data used in our analysis, so that researchers can re-run our analysis for validity.

VIII. Conclusion

Developers spend significant time not only to detect flaky tests but to identify the root causes of test flakiness. Researchers and practitioners actively working to support developers and testers to mitigate test flakiness. In this paper, we proposed a light weight technique named *trace-back coverage* to detect flaky tests. Furthermore, *trace-back coverage* was combined with other factors (multi-factor) such as test smells inducing test flakiness, flaky frequency and test case size to investigate the effect on revealing test flakiness. Results show that combining all factors often yields high precision, accuracy and recall (0.78–0.98). In addition, We concluded that aggregated information about these fours factors provides significant help to developers about the root causes of test flakiness. Developers can use these information as base-line to investigate further. *MDFlaker* as compared to other tools has its advantages such as less effort to implement for any given language, less time for execution, and providing additional information to testers for root cause analysis. However, *MDFlaker* provides higher accuracy as compared to Bayesian Network but less accuracy as compared to DeFlaker.

References

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

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

[3] C. Ziftci and D. Cavalcanti, "De-Flake Your Tests : Automatically Locating Root Causes of Flaky Tests in Code At Google," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2020, pp. 736–745, iSSN: 2576-3148.

[4] D. Silva, L. Teixeira, and M. d'Amorim, "Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2020, pp. 301–311, iSSN: 2576-3148.

[5] K. Presler-Marshall, E. Horton, S. Heckman, and K. Stolee, "Wait, Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness," in *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, May 2019, pp. 7–13.

[6] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2020, pp. 403–413, iSSN: 2332-6549.

[7] A. Groce and J. Holmes, "Practical Automatic Lightweight Nondeterminism and Flaky Test Detection and Debugging for Python," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2020, pp. 188–195.

[8] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and Ranking Flaky Tests at Apple," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Oct. 2020, pp. 110–119.

[9] J. Micco, "Flaky Tests at Google and How We Mitigate Them, https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. Accessed [2019-04-15 18:48:16]." [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[10] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 433–444.

[11] S. Liviu, "A machine learning solution for detecting and mitigating flaky tests," Oct. 2019. [Online]. Available: https://medium.com/fitbit-tech-blog/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests-c5626ca7e853

[12] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a Bayesian Network Model for Predicting Flaky Automated Tests," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. Lisbon: IEEE Comput. Soc, Jul. 2018, pp. 100–107.

[13] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653, event-place: Hong Kong, China. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[14] A. Sjöbom, *Studying Test Flakiness in Python Projects : Original Findings for Machine Learning*, 2019. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-264459

[15] S. Thorve, C. Sreshtha, and N. Meng, "An Empirical Study of Flaky Tests in Android Apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 534–538.

[16] A. Ahmad, O. Leifler, and K. Sandahl, "Empirical Analysis of Factors and their Effect on Test Flakiness - Practitioners' Perceptions," *arXiv:1906.00673 [cs]*, Jun. 2019, arXiv: 1906.00673. [Online]. Available: http://arxiv.org/abs/1906.00673

[17] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pp. 830–840, 2019, arXiv: 1907.01466. [Online]. Available: http://arxiv.org/abs/1907.01466

[18] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 101–111. [Online]. Available: https://doi.org/10.1145/3293882.3330570

[19] A. Bertolino, B. Miranda, and R. Verdecchia, "Know your neighbor: fast static prediction of test flakiness," *ISTI Technical Reports*, vol. 2020, no. 001, p. 1, Jan. 2020. [Online]. Available: https://doi.org/10.32079/ISTI-TR-2020/001

[20] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the Vocabulary of Flaky Tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 492–502. [Online]. Available: http://doi.org/10.1145/3379597.3387482

[21] A. Ahmad, O. Leifler, and K. Sandahl, "An Evaluation of Machine Learning Methods for Predicting Flaky Tests," in *8th International Workshop on Quantitative Approaches to Software Quality in conjunction with the 27th Asia-Pacifc SoftwareEngineering Conference (APSEC 2020) Singapore*, Dec. 2020, pp. 37–46.

[22] F. Palomba and A. Zaidman, "The smell of fear: on the relation between test smells and flaky tests," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2907–2946, Oct. 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09683-z

[23] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992, publisher: [American Statistical Association, Taylor & Francis, Ltd.]. [Online]. Available: https://www.jstor.org/stable/2685209

[24] F. G. de Oliveira Neto, A. Ahmad, O. Leifler, K. Sandahl, and E. Enoiu, "Improving Continuous Integration with Similarity-based Test Case Selection," in *Proceedings of the 13th International Workshop on Automation of Software Test*, ser. AST '18. New York, NY, USA: ACM, 2018, pp. 39–45. [Online]. Available: http://doi.acm.org/10.1145/3194733.3194744

[25] J. Percival and N. Harrison, "Developer Perceptions of Process Desirability: Test Driven Development and Cleanroom Compared," in *2013 46th Hawaii International Conference on System Sciences*, Jan. 2013, pp. 4800–4809.