FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment

Maxime Cordy
SnT, University of Luxembourg
Luxembourg
maxime.cordy@uni.lu

Renaud Rwemalika
SnT, University of Luxembourg
Luxembourg
renaud.rwemalika@uni.lu

Adriano Franci
SnT, University of Luxembourg
Luxembourg
adriano.franci@uni.lu

Mike Papadakis
SnT, University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

Mark Harman
Meta Platforms inc., and University College London, United Kingdom
mark.harman@ucl.ac.uk

ABSTRACT

Much research on software testing makes an implicit assumption that test failures are deterministic such that they always witness the presence of the same defects. However, this assumption is not always true because some test failures are due to so-called flaky tests, i.e., tests with non-deterministic outcomes. To help testing researchers better investigate flakiness, we introduce a test flakiness assessment and experimentation platform, called FlakiMe. FlakiMe supports the seeding of a (controllable) degree of flakiness into the behaviour of a given test suite. Thereby, FlakiMe equips researchers with ways to investigate the impact of test flakiness on their techniques under laboratory-controlled conditions. To demonstrate the application of FlakiMe, we use it to assess the impact of flakiness on mutation testing and program repair (the PRAPR and ARJA methods). These results indicate that a 10% flakiness is sufficient to affect the mutation score, but the effect size is modest (2% - 5%), while it reduces the number of patches produced for repair by 20% up to 100% of repair problems; a devastating impact on this application of testing. Our experiments with FlakiMe demonstrate that flakiness affects different testing applications in very different ways, thereby motivating the need for a laboratory-controllable flakiness impact assessment platform and approach such as FlakiMe.

ACM Reference Format:
Maxime Cordy, Renaud Rwemalika, Adriano Franci, Mike Papadakis, and Mark Harman. 2022. FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510194

1 INTRODUCTION

Test flakiness is the property of a test case and system under test, that the test can pass on one occasion, yet fail on another, without the tester changing anything other than the fact that the test is executed on two different occasions. This behaviour has a number of causes, such as non-determinism in the system under test, instability in the infrastructure that provides the test environment, and variability in the results produced by services and components upon which the system under test depends.

Flakiness has a profound impact on all applications of software testing because it increases test signal uncertainty; the tester can never be sure that a failure is genuine and this may waste effort (investigating false positives) or lose important signals (from switching off flaky tests). Companies such as Google and Facebook have highlighted the problem of test flakiness [14, 20, 26, 30], indicating that it is one of their primary concerns for software testing. Some companies have also launched specific challenges to the research community to tackle this problem [15].

Flakiness impacts each form of testing in different ways. E.g. in mutation testing, the mutation score will vary, dependent on flakiness, confounding this variability with the influence of the quality of the test that the score seeks to assess. In automated program repair, the certainty we have that a repair is correct will be affected by flakiness, as will be the ability of the repair technique to localise the point at which to attempt a patch. Indeed, it has been argued that *all* forms of testing need to be reformulated to take account of flakiness in order to find techniques that can cope well in the presence of unavoidable flakiness [2, 14, 23]. This means that testing techniques need to be re-investigated under flakiness conditions to assess their robustness on varying degrees of flakiness.

In order to address the problems posed by flakiness, researchers need ways to investigate the impact of flakiness. Naturally, studies should be conducted on real-world systems to explore this impact [26, 29]. However, researchers also need the ability to experiment with flakiness in laboratory-controlled conditions. Such laboratory control would circumvent the limited number, availability, and reproducibility of flaky test datasets, and allow researchers to report results on the impact of *varying degrees* of flakiness on the test techniques they propose and introduce.

To address this need, we introduce FlakiMe, a tool that allows researchers to seed a controlled degree of flakiness into a given test suite and system under test. FlakiMe equips researchers with a laboratory-controllable environment in which they can simulate a rich set of flakiness scenarios and conditions. Specifically, FlakiMe can:

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510194

- be seamlessly integrated into any Java project as a Maven plugin, without requiring any modification to the code of the project;
- trigger flaky failures during test executions, at any point during these executions;
- rely on user-defined flakiness prediction models to associate tests with a likelihood to be flaky;
- control the degree of injected flakiness via a parameter that defines the probability threshold above which execution of any flaky test results in a flaky failure.

This paper introduces FlakiMe and illustrates its application to the assessment of flakiness impact on mutation testing – implemented in PIT [6] – and "generate and validate" automated program repair (G&V APR) – more precisely on the PRAPR [11] and ARJA [43] repair methods.[1]

To demonstrate the capability and the usefulness of FlakiMe, we combine it with a state-of-the-art flakiness prediction approach [5, 13, 34] that identifies tests that could be flaky based on their code similarity to real flaky tests. That is, the approach predicts the probability that a test is flaky based on the terms (tokens) this test contains. Previous studies have demonstrated that these vocabulary-based approaches can effectively predict flaky tests, with an F1 score greater than 90%.

In FlakiMe, we use this flakiness prediction approach to build a theoretical model of flakiness that identifies candidate tests to inject realistic flakiness into. We, therefore, obtain the following benefits:

- Each test has a different probability to trigger a flaky failure.
- Tests that execute similar pieces of code have a dependent probability to trigger a flaky failure.
- Flaky failures can occur at any location in the tests and this occurrence depends on the code of the test.

There is, to date, no evidence that this model matches the actual behaviour of real flaky tests and, for this reason, experimental results obtained with FlakiMe may differ from results that would be obtained with real flaky tests. FlakiMe enables laboratory-controlled experimentation with extremes, and therefore it is not directly concerned with the specific settings/characteristics of a particular environment context. As such, we give experimental controllability over the degree of flakiness, so that researchers can stress-test their techniques and study their limits. It remains important to validate research techniques in practice, through empirical studies with real flaky tests. What FlakiMe offers is the capacity to conduct complementary experimental analyses under broader and more controllable settings than what real-but-rare flaky tests enable.

Previous research on flakiness has helped to identify the main causes of flakiness [27, 29], and has introduced techniques to either reduce or ameliorate its effects. However, hitherto, no systematic way of evaluating the sensitivity of arbitrary software testing problems to flakiness has been introduced. We fill this gap and report results on the use of our FlakiMe platform to yield insights on two software testing problems. Thus, our key contribution is the evidence that the use of FlakiMe leads to interesting and actionable results. Specifically, we use FlakiMe to perform two laboratory-controlled experiments on:

1. **Mutation testing**: We show how FlakiMe allows us to investigate the *size* of this effect. In particular, using FlakiMe we can reveal that:
   (a) A small amount of flakiness can affect the mutation score (10% of flakiness failures yields mutation score variations between 2% to 5%);
   (b) When the degree of flakiness increases, the mutation score follows an asymptotic growth where additional increases in the degree of flakiness have a rapidly diminishing impact on mutation testing.
   (c) Taken together with the above findings we form a take-home message for mutation testing researchers: flakiness is a potential problem but with limited effect. Although researchers should always take into account the flakiness effects on the mutation scores they report, the results suggest that it is not sufficient to "poison the well".
2. **Automated program repair**: For the application of repair, we found that the impact of flakiness is stronger than it is on mutation testing. Specifically, we show that the same degree of flakiness has different effects on different systems. This indicates that research on automated repair needs to analyze how sensitive to flakiness their test suites and subjects are. Our results show that FlakiMe can be used to pre-select suitable subjects and to validate the key decisions made by the studied techniques. More precisely, we show that:
   (a) Deterministic repair is increasingly affected by the number of tests covering the produced patches. In our studied projects, only 50% of patches remain proposed by the techniques as a result of flakiness.
   (b) Non-deterministic repair experiences a drop in the number of patches produced by 60% to 100%, with the worst case (total failure; 100% drop) occurring for 36% of the programs studied. Exploiting knowledge about the non-flaky failures mitigates this effect, yielding between 2 to 600 times more patches and even allowing the repair of cases that were before hindered by flakiness.

2 RELATED WORK

Previous work on test flakiness [1, 8, 9, 20, 21, 35] has primarily focused on identifying its causes. Luo *et al.* [27] were the first to propose a formal classification of the root causes of test flakiness. Relying on this definition, recent studies [1, 4, 18–21, 26] highlight various degrees of flakiness in industrial code bases (e.g. 4% of flaky failures at Google [26]). Other studies have been conducted addressing non-deterministic system such as machine learning models [8], different programming languages [12] or automatically generated tests [33]. The follow-up work aimed at the automated identification of flaky tests [10] and the development of tools that identify and/or remove flaky tests, such as iFixFlakies [38], iDFlakies [22], RootFinder [20], DeFlaker [4], SHAKER [39], FLASH[8] or the work of Malm *et al.* [28] which proposes to insert delays in tests to increase their robustness.

The presence of flaky tests degrades the effectiveness of test suites. Therefore, flaky tests should be fixed/controlled early, ideally as soon as flakiness is introduced [32]. Facing the need to detect

[1] We will refer to G&V APR simply as "automated program repair".

flaky behavior before it manifests itself, researchers started investigating flakiness prediction models based on supervised learning. King *et al.* [17] rely on a Bayesian network model defined over code metrics related to flakiness. Similarly, FLAST[41] uses code metrics to feed a K-nearest neighbours model. Recent work has shifted from code metrics to vocabulary-based approaches. Pinto *et al.* [34], followed by Haben *et al.* [13] and Camara *et al.* [5], developed random forest models that learn from test code tokens in order to predict new flaky tests based on how similar their vocabulary is to known flaky tests. Finally, approaches like FlakeFlagger [3] and the classifier presented by Lampel *et al.* [24] rely on dynamic features (observed at runtime) instead of the static features used in the other studies. With FlakiMe, researchers have now the ability to leverage such flakiness prediction models in order to control the degree of flakiness present in test suites and assess the sensitivity of testing techniques to this flakiness.

The inherent non-determinism of flaky tests, combined with the specific environmental conditions under which flakiness manifests itself, makes it inherently difficult to collect flaky test datasets and even more difficult to consistently reproduce flaky failures. As a result, few studies have analysed the effects of flakiness on software testing techniques.

Mutation testing. Shi *et al.* [37] evaluated the effects that non-deterministic coverage (in test executions) can have on mutation score and proposed a way to reduce this problem. The study shows that the mutation score can vary up to 5% when ignoring non-determinism. These findings are based on *in vivo* observations of *non-determinism* in 30 open-source software projects. None of the 30 projects they studied exhibited flaky behavior in the test outcome, only in the test coverage. This highlights the difficulties to construct flaky test datasets suitable for research and to reproduce real flakiness occurrences. Therefore, we complement such studies with FlakiMe and its capability to produce *in vitro* laboratory-controlled flakiness. With FlakiMe, we simulate test failures with different flake rates and observe the asymptotic growth in mutation score as the flake rate increases. Our results corroborate Shi et al.'s general finding that a small amount of non-determinism can affect the mutation score. Our novel finding is that the effect of flakiness saturates rapidly when the flakiness rate increases.

Automatic Program repair. Qin *et al.* [36] have analyzed the impact of specific causes of flakiness on APR. Like our study, they rely on the Defects4J dataset as a source of buggy programs – the most established dataset in APR. However, their analysis is limited to flaky failures caused by the use of different JDK versions. Their results show that a few flaky failures (0.3% of all test executions) negatively impact the suspicious statement localization in more than 20% of the programs and reduce the repair capabilities of the APR tool by up to 87%. By contrast, our study introduces a broader set of flakiness instances in Defect4J tests based on how similar their code is to flaky tests observed in the field. We investigate in depth how the rate of occurrence of flaky failures affects each step of the APR process, from fault localization to patch validation. While Qin et al. [36] observe negative effects, these effects are somehow limited (APR can still fix bugs). Our novel finding is that scarce flakiness occurrences can have a profound effect and even annihilate the capacity of APR methods. We also identify the key components that are affected and suggest mitigation strategies.

Fault Localization. Vancsics *et al.* [40] have studied the effects of flakiness on fault localization. They seeded flakiness in tests suites by making tests randomly flake (with a uniform distribution), whereas we use calibrated prediction models from the literature and a varying flakiness rate in order to observe trends and extremes. Their observations are aligned with ours: flakiness has a significant impact and falsely alters the ranking of suspicious statements. They also show that different ranking formulae are impacted differently. In our study, we do not observe the effects of flakiness on fault localization in isolation but within the broader use case of APR. We further demonstrate that the impact of flakiness on fault localization is the main factor explaining the reduced effectiveness of APR and propose mitigation strategies accordingly. Finally, our platform is generic and enables experimentation on different testing techniques, including – but not limited to – FL.

Test Selection/Prioritization. Leong *et al.* [26] has reported that flaky tests significantly mislead the test selection algorithms used in Google's continuous integration environment. Lam *et al.* [23] have conducted a focused study on the impact of test order dependency, a specific form of flakiness. Though our study focuses on mutation testing and APR, FlakiMe can also support sensitivity studies on other testing techniques through the seeding of varying degrees of flakiness.

3 FLAKIME

FlakiMe[2] is a flexible tool through which researchers can simulate a rich set of scenarios and conditions for flakiness. It is distributed as a Maven plugin. Hence, testers can seamlessly integrate FlakiMe into any Java project as part of the Maven configuration, without requiring any modification to the code of the project or its environment.

FlakiMe instruments the bytecode of the tests by introducing a payload that can trigger flaky failures at any test execution point, *i.e.* flake points. FlakiMe, therefore, introduces test failures but does not make some tests incorrectly pass.[3] These test failures are introduced at specific test code locations named *flake points*. Concretely, a flake point can be any location that corresponds to the end of a basic control flow block (i.e. a sequence of contiguous instructions that does not contain jump/branch instructions except the last one). The probability that a flaky failure occurs at a specific flake point depends on (1) a user-defined *flakiness prediction model* and (2) a real-valued parameter named the *nominal flake rate*. FlakiMe implements a test instrumenter that injects probable program failures at any flake point during test execution. To simulate a flaky failure, FlakiMe raises an unchecked exception, thereby causing the test to fail. The exception is guarded by a probabilistic condition which depends on the probability of the test execution to flake at this point.

To qualify the probabilistic condition that guards the unchecked exception, we rely on three concepts:

- The *test flakiness probability*, $P_{flakiness} \in [0, 1]$, represents the likelihood that some (part of) test code is flaky. This value is determined by the flakiness prediction model integrated into FlakiMe.

[2] https://github.com/serval-uni-lu/flakime
[3] We will consider extending FlakiMe with this capability in the future.

- The *nominal flake rate*, $FR_n \in [0, 1]$, represents the probability that the execution of a flaky test results in a flaky failure. This rate is a parameter that researchers can set and control in their experiments to conduct sensitivity analyses.
- The *effective flake rate*, $FR_e \in [0, 1]$, is the actual probability for the execution of any test to result in a flaky failure. It is defined as $FR_e = FR_n \times P_{flakiness}$. Therefore, the higher the user sets the *nominal flake rate*, the higher number of flaky failures this user will observe.

In addition to the nominal flake rate that FlakiMe users control entirely, the users can easily plug in their own flakiness prediction model. For instance, such models can learn from previously observed flaky tests to predict the likelihood that unseen test instances are flaky [3, 5, 13, 17, 34]. FlakiMe can benefit from studies characterizing flakiness to build specific environmental perturbations that mimic real-world flakiness as it has been observed in the field. There currently exists no perfect model of real flaky tests in the literature. Hence, the realism of the flakiness that FlakiMe injects is bound to the evidence that the integrated model matches real flaky tests. Hence, researchers should use FlakiMe with the model that best reflects their environmental settings and working assumptions, and be aware that the generality of their conclusions is limited to the validity of these working assumptions in the real world. We discuss this further in Section 7 – Threats to Validity.

In our study, we take advantage of a generic state-of-the-art approach that predicts which tests could be flaky based on their test code tokens similarity with previously observed real flaky tests [5, 13, 34]. The advantage of this model is that it applies to pieces of code of any length. It can, therefore, be used to define dependent probabilities on multiple flake points of the same test. Also, the model being based on code vocabulary, tests that execute similar pieces of code have a dependent likelihood to be flaky.

More specifically, the approach of Pinto et al. [34] uses a set of 1,874 flaky tests extracted from 24 open-source projects [4] to train a classifier. They represent a training sample with its binary label (test being flaky or not) and its token feature vector, i.e.: for each token contained in any of the 1,874 flaky tests, whether the token is present. To compute this vector, the approach tokenises the code of the test, removes a predefined list of stop words, and builds a Boolean vector such that each element in the vector indicates the presence or absence of the corresponding token in the test. With this encoding, Pinto et al. trained a supervised classification model to classify tests as flaky or not, thus based on the terms (tokens) this test contains. Among the all classification models they have considered in their empirical evaluation, Pinto et al. have demonstrated that a random forest classifier yielded the best prediction performance (the highest precision and F1 score).

In FlakiMe, we leverage the approach of Pinto et al. (that predicts the probability that a test is flaky) in order to determine the flakiness probability $P_{flakiness}$ of a test based on the terms it contains. To use this approach in our study, we have extended the initial training set with the token of our test subjects (i.e., the project under flakiness simulation) and we trained from scratch the random forest following Pinto et al.'s protocol [34]. Then, we use the model to determine the test flakiness probability at any flake points in the tests of the project under study. More precisely, we use the model to compute the probability that a given test is flaky. We, then, spread the probability mass over all flake points in the test, based on the vocabulary of the code that precedes each flake point. Hence, flake points lying at statements found in many flaky tests will receive a higher probability than flake points that include no risky operations.

Figure 1 shows the distribution of the test flakiness probability that we obtain for our studied projects (see Section 5.4. We observe that the shape of the distribution changes for each project. When compared to `Chart` (Figure 1a) the project `Time` (Figure 1d) has a median *flakiness probability* of 0.45 while the former has a median value of 0.19. `Time` exhibits less variance across tests and is, on average, more prone to flakiness than `Chart`. It is noteworthy that these probability values do not correspond to the actual probability that tests executions flake, but they rather serve as a means of distinguishing the inherent proneness of different tests to be flaky. FlakiMe weights each of these probability values with the nominal flake rate, thereby allowing users to control the effective flake rate.

FlakiMe is designed to be an extensible tool that can be used by researchers to test novel methods under different assumptions. Thus, we design FlakiMe so that it can easily incorporate new flakiness prediction models to account for such assumptions. To do so, we integrate the concept of *Flakikess Model* which is an abstract Java class that can be extended to easily integrate additional models. This abstract class contains three methods: (1) `preProcess` which is run before the instrumentation and provides information about the tests to be instrumented; (2) `getEffectiveFlakeRate` which returns the *effective flake rate* given a test method and a line number, and (3) `postProcess` which allow for eventual cleaning or any other post-processing operation.

To ease the integration of other methods/models, the framework provides static information about the test code and an interface to integrate modules assigning the flakiness probabilities.

4 THE IMPACT OF FLAKINESS ON SOFTWARE TESTING TECHNIQUES

4.1 Mutation Testing

Mutation testing [31] determines the proportion of mutants (artificially injected defects) causing tests to transition from passing to failing. Starting from a test suite $t_1, \ldots, t_n$ that passes on the original program P, mutation testing generates mutants $M_1, \ldots, M_k$ by altering the syntax of P and, then, runs the test suite on the mutants. The test suite $t_1, \ldots, t_n$ kills a mutant M_j if the execution of at least one test on M_j fails. The mutation score (number of mutants killed divided by the number of mutants) is a frequently used metric for measuring test thoroughness [31]. One can see this as an $n \times k$ matrix, where each cell is related to a test t_i and mutant M_j pair, and denotes whether t_i killed M_j or not.

In the absence of flakiness, such a matrix is determined by the tests and the mutants. However, in the presence of flakiness things change arbitrarily; a flaky test that passes on the original program can fail on a mutant, leading to a kill. Thus, running the test suite with FlakiMe results in swapping the status of some mutants. The probability of swapping, in this case, is equivalent to the effective probability that the concerned tests flake. Hence, failed flaky

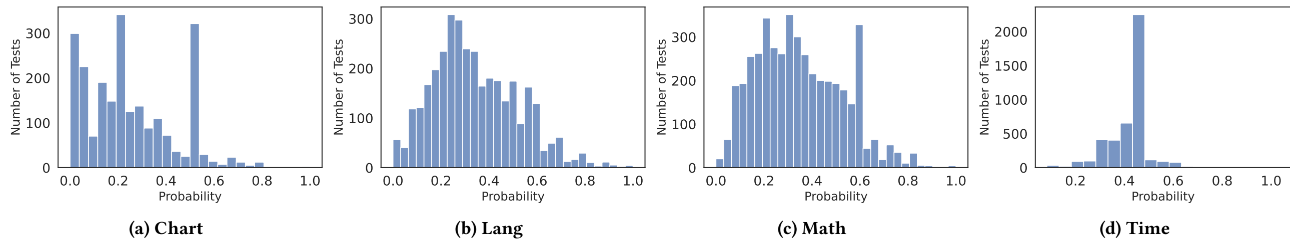


Figure 1: Distribution of test flakiness probability assigned by the vocabulary model employed in FlakiMe to each test subject.

test executions artificially inflate the mutation score, causing an overestimation of the test suite's fault revealing potential.

4.2 Program Repair

Automated program repair aims at generating patches (modifications of the software code that fix bugs) for programs with bugs witnessed by failing test cases. In this line of work, effectiveness is measured by the number of valid patches, generated within a given time limit. A valid patch is one that compiles and passes all tests, including the initially failing tests (that witnessed the bug).

Since the validity of patches is determined by the test case results, it is interesting to see the extent to which flakiness can impact patch selection. In other words, we would like to check the sensitivity of repair methods to noisy signals caused by flakiness. FlakiMe impacts this validity check by making a test fail randomly. In the repair process, such a patch could be mistakenly discarded.

An increasing number of repair methods have appeared over the years [7]. We select two recent methods that exhibit fundamental differences of approach for greater diversity. These are PRActical Program Repair (PRAPR) [11] and Automated Repair for Java Programs (ARJA) [43]. PRAPR uses mutation testing to generate patches while ARJA uses genetic programming. Another key difference between the two approaches lies in their usage of Fault Localization (FL) [42]. Where PRAPR only uses FL to prioritise the order in which the patches are generated, ARJA uses it to identify the most suspicious statements to repair.

4.2.1 Deterministic mutation-based repair (PRAPR). Unlike ARJA, PRAPR requires the user to specify the failing test(s) that witness the bug that is to be repaired. Then, it applies Fault Localization (FL) to associate an estimated degree of suspiciousness to the statements covered by the failing tests. The most suspicious statements are targeted first, so as to increase the likelihood of finding a good fixing point early. However, all statements are investigated given sufficient time. To repair the program, PRAPR applies a predefined set of mutation operators on each of the covered statements. More suspicious statements are mutated first. This process results in a deterministic set of patches defined by the mutation operators and the mutable locations. To validate the patches, only tests covering the mutated statements as well as the tests given as input are executed against the mutants. The patch candidates (mutant programs) for which all tests pass constitute the resulting set of valid patches.

4.2.2 Genetic programming-based repair (ARJA). ARJA is a GenProg-like [25] tool. It generates a population of patches that evolve over a predefined number of generations. ARJA does not ask the user to specify the initially failing tests. Instead, it runs the entire test suite, applies FL, and considers *all* tests that failed to retrieve the set of suspicious statements. Then, it discards the statements with suspiciousness values below a predefined threshold and collects the statements that (i) are covered by at least one test covering the suspicious statements (ii) have some dependency with the suspicious statements. A patch is formed by altering these statements. ARJA uses the NSGA-II genetic algorithm to evolve the patches using a fixed-size population and by producing patches for a fixed number of evaluations. To evaluate a patch, ARJA runs the initially-failing tests and all other (passing) tests that cover suspicious statements (not discarded during filtering). If no tests fail, including the initially failing ones, the patch is considered valid.

ARJA generates the same number of candidate patches over different runs. However, the patches will differ due to the randomness in the evolution of the population. Hence, the number and content of valid patches vary from one run to another.

Flaky tests may affect the initial test suite run, impacting the FL estimates (suspiciousness scores). This can have a double effect: change the patch search space and alter the number of tests to be used for patch validity check. Variations in these numbers provide a coarse view of the extent to which the use of FlakiMe has reshaped the search space and affects the likelihood of finding a valid patch. During patch validation, flaky test failures might occur and add noise to the signal. Not only does this cause the algorithm to discard valid patches but it also perturbs its learning, potentially hindering its effectiveness.

4.3 Suspicious Statement Selection

Suspicious statement selection in many repair techniques – including ARJA – relies on Spectrum-Base Fault Localization (SBFL). Given a set of statements $\{s_1, \ldots, s_s\}$ and a test suite $\{t_1, \ldots, t_n\}$, SBFL assigns a suspiciousness score to each statement based on the number of failing and passing tests covering them. It does this by building an $n \times s$ matrix where each cell records whether a particular test covers a particular statement. Then, it runs all tests and keeps a record of those that pass and those that fail. Based on this, it computes a suspiciousness score for every statement. For instance, Ochiai, the metric used by PRAPR and ARJA, assigns to any statement s the score: $s_f / \sqrt{(s_f + n_f) \cdot (s_f + s_p)}$ where s_f is the number of failing tests covering the statement s, n_f is the number of failing tests, and s_p is the number of passing tests covering s.

Compared to a non-flaky test suite with clearly identified failing tests, FlakiMe affects the Ochiai score of all statements because tests

sometimes fail instead of pass. This increases the values n_f and s_f (if the failing flaky tests cover s). Thus, flakiness can either increase or decrease the Ochiai score of the statements. When a surrounding repair method discards statements based on their suspiciousness (as in the case of ARJA), such differences can largely affect the search space and reduce the effectiveness of the repair.

5 EXPERIMENTAL SETUP

5.1 Research Questions

Our first question concerns mutation testing and examines (quantitatively) the extent to which mutation score can be inflated by test flakiness:

RQ1 *How much does flakiness artificially inflate the mutation score of given test suites?*

To answer this question we check the effect of flakiness on the mutation scores of the whole projects' test suites first, then on randomly chosen samples. Our interest is on the divergence of those scores under different degrees of flakiness.

In automated program repair, the observable effect of flakiness is to reduce the number of generated valid patches. We want to check the sensitivity of repair methods on flakiness. Thus, we ask:

RQ2 *How sensitive to flakiness is the effectiveness of program repair at generating valid patches?*

We study this question on two state-of-the-art automatic program repair tools leveraging different strategies, namely, PRAPR and ARJA. While our goal is not a comparison of the tools, we aim to highlight the ways flakiness affects different approaches and, from there, to suggest mitigation strategies. Thus, we divide RQ2 into the following subquestions:

How much does flakiness decrease the number of valid patches produced by deterministic mutations?

How much does flakiness decrease the number of valid patches produced by genetic programming?

Going a step further, we also investigate the way PRAPR and ARJA use Spectrum-Based Fault Localization (SBFL). PRAPR applies SBFL only as a prioritisation step to minimise the time to generate the first valid patch. Ultimately, it considers only the suspicious statements covered by the failing tests that the user-provided. Conversely, ARJA uses SBFL to define its search space (discarding less suspicious statements). This has three consequences: (a) the search space encompasses more candidate patches that do not fix the bug (since they target wrong statements), reducing de facto the effectiveness of ARJA; (b) the faulty statement may be removed from the search space, making it impossible to generate a valid patch; (c) the number of tests – both failing and passing – executed to validate candidate patches is increased. Consequently, we also investigate the scenario where no flaky failures occur during SBFL. In this case, the suspicious statement search space is not compromised which could help to alleviate the effects of flakiness. To this end, we seek to answer:

RQ3 *Does making fault localization target real failing tests improve the robustness of program repair against flakiness?*

To complement our analysis, we study the sensitivity of SBFL to flakiness with respect to the selection of suspicious statements. We aim to measure the number of faulty statements that remain selected as the flakiness rate increases, and the number of non-faulty statements that are kept out of the search space. Hence, we are interested at:

RQ4 *How much does flakiness hinder the ability of threshold-based suspicious statement selection?*

Overall, the aim of our study is to demonstrate that FlakiMe can yield interesting insights on the techniques' behaviours and robustness. Our goal is to show that the sensitivity of some decisions and method characteristics to flaky tests can be exposed through the lens of FlakiMe, which paves the way for the study and design of mitigation strategies.

5.2 Nominal Flake Rate

The nominal flake rate FR_n is the parameter that we can set in our experiments to control the effective rate FR_e at which flaky failures occur. In practical applications, FR_e depends on environmental factors and conditions that vary significantly and is, therefore, hard to generalize outside a particular context. For example, industrial evidence shows the existence of test suites involving 15%-45% flaky tests [1, 4, 18–21, 26], which differs significantly from open-source projects [3]. However, these studies do not report on the rate at which these tests flake. Because of this, we consider it essential and insightful to experimentally analyse the sensitivity of testing techniques to varying degrees of flakiness. We, therefore, consider a range of FR_n from 0.01 to 0.50 (we stop at 0.20 for ARJA because this repair method could not produce patches above this rate).

5.3 Third-Party Tools

To study the effect of flakiness on mutation testing, we use the open-source tool PIT [6], with its default operator set. The mutation score MS of a test suite T on a program P can be expressed as: $MS(P,T) = |K|/(|M| - |E|)$ where $|K|$ is the number of mutants killed, $|M|$ is the total number of mutants and $|E|$ is the number of equivalent mutants. We ignore the analysis of equivalent mutants since they do not impact our analysis and use a simplified mutation score measure: $\overline{MS}(P,T) = |K|/|M|$ which is independent of the equivalent mutants. Thus, as flakiness is introduced, only the number of killed mutants $|K|$ influences the mutation score.

To analyze the sensitivity of automated program repair, we select two tools leveraging different strategies, PRAPR and ARJA. PRAPR is available as a Maven plugin and as a Docker image[4]. We use the Maven plugin in conjunction with the FlakiMe plugin to perform our experiment.

ARJA is retrieved from GitHub[5] and modified to print additional statistics related to its execution and notify the system which step it is performing (SBFL or patch validation).

To account for random variations introduced by flakiness and by the patch generation process in the case of ARJA, we execute 10 runs for each degree of flakiness. This number of repetitions is a compromise between statistical relevance and computation cost.

[4]https://github.com/prapr/prapr
[5]https://github.com/yyxhdy/arja

5.4 Test Subjects

Defects4J [16] is a set of real bugs harvested from Java projects. It is one of the most popular datasets in evaluating automatic program repair techniques, including PRAPR and ARJA. In our study, we consider the bugs for which the techniques succeeded. An important success metric here is the ability of the techniques to generate genuine patches (semantically equivalent to the developers' patch). Thus, for PRAPR, we picked the 16 buggy versions from five projects originating from the Defects4J dataset for which PRAPR produced at least one genuine patch. We discarded the buggy programs for the Closure project because PRAPR requires more than 64GB of RAM to repair them [11]. For ARJA, we consider 11 buggy versions from two programs of Defects4J, for which the tool generated at least one genuine patch (reported in ARJA's supplementary material[6]) and for which we could successfully generate valid patches (using the default settings of the tool). Unfortunately, we could not generate valid patches for some programs, due to differences in the tool configurations and/or infrastructures. Nevertheless, to increase diversity, we also considered 3 projects for which ARJA could generate valid (but not necessarily genuine) patches , i.e. patches that pass all tests.

For mutation testing, we consider the latest releases (non-buggy) of four projects that we use in the repair experiments: time, lang, charts and math. We choose these projects to maintain a certain consistency across our experiments.

6 RESULTS

6.1 RQ1: Mutation Testing

We investigate the effect of flakiness on the mutation score by injecting flaky failures with a nominal flake rate ranging from 0.0 to 0.5. We repeat the experiment 10 times (for each rate).

Figure 2a shows the average mutation score overall runs. We observe that mutation score increases more steeply at lower flakiness rates. This indicates that flaky tests, even at a small degree, are sufficient to introduce noise, although the effect remains modest overall. One reason leading to this fast increase comes from tests with high flakiness probability. These tests tend to fail as soon as flakiness is introduced. This effect is project-dependent and bounded by the number of mutants that survive and are covered by the flaky tests – hence the asymptotic behaviour. However, the total increase of mutation score remains low even when the degree of flakiness increases. For instance, when increasing the nominal flake rate from 0.0 to 0.5, the mutation score of `Time` raises from 31.73% to up to 38.93% (increase of 7.69%) while the score of `Lang` increases from 85.58% to up to 90.57% (increase of 4.99%).

Figure 2b shows the standard deviation of the mutation score as the nominal flake rate increases. Without flakiness, the outcome of the tests is deterministic and, therefore, we observe a standard deviation of zero. When flakiness occurs, the standard deviation remains low with average values ranging from 0.01 to 0.21.

Figure 2c allows us to better observe the difference in mutation score that each nominal flake rate induces. We see that the median values for a nominal flake rate of 0.1 range between 1.86% (Lang) and 4.78% (Time). The values for a nominal flake rate at 0.5 ranges from 4.11% (Lang) to 7.87% (Time). We observe that the smallest degree of flakiness already results in about half of the total increase of mutation score (i.e. at 0.5 nominal flake rate) and that mutation score plateaus from thereon.

[6]https://github.com/yyxhdy/arja-supplemental/blob/master/arja-supplemental.pdf

Table 1: Impact of flakiness on the valid patches generated by PRAPR. $|P|$ and $|V|$ denote the number of (all) patches and valid patches originally generated by PRAPR. $|T_v|$ is the average number of tests covering a valid patch. $|V_f|$ is the average number of valid patches in the flaky case, where $f \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ is the nominal flake rate.

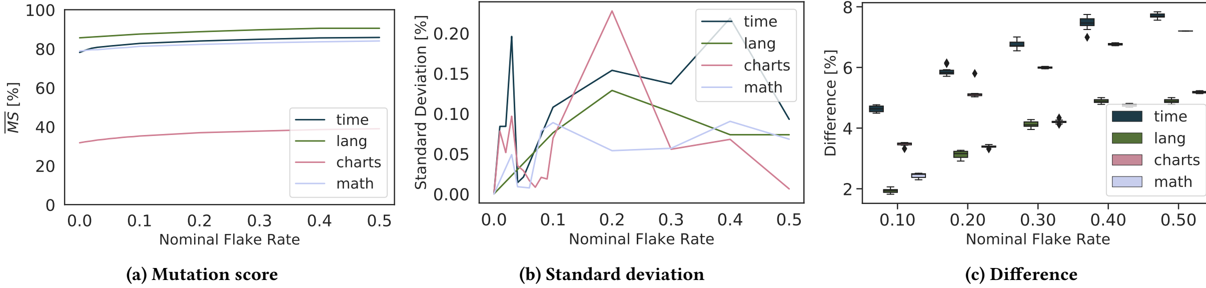
Bug	$\|P\|$	$\|V\|$	$\|T_v\|$	$\|V_{0.1}\|$	$\|V_{0.2}\|$	$\|V_{0.3}\|$	$\|V_{0.4}\|$	$\|V_{0.5}\|$
math-5	211	3	36.3	2.4	1.7	1.0	0.8	0.6
math-34	146	1	1.0	1.0	0.5	0.2	0.0	0.0
math-50	931	40	3.1	32.9	27.2	25.1	22.2	20.3
math-82	2017	7	14.0	5.5	4.4	2.7	1.8	0.8
math-85	1190	4	13.0	2.4	1.2	0.7	0.1	0.0
time-11	2951	36	6.1	28.0	21.8	19.3	15.2	13.3
time-19	3423	2	705.0	0.4	0.2	0.0	0.0	0.0
lang-6	206	1	31.0	0.4	0.1	0.0	0.0	0.0
mock.-29	2842	2	5.0	1.2	0.9	0.5	0.4	0.0
mock.-38	482	3	77.7	1.2	1.0	0.4	0.4	0.1
chart-1	3721	1	38.0	0.9	0.6	0.6	0.4	0.2
chart-11	158	2	16.0	1.4	0.9	0.7	0.3	0.1
chart-12	2245	2	3.5	1.2	0.9	0.4	0.0	0.0
chart-20	240	1	95.0	0.4	0.4	0.3	0.1	0.1
chart-24	133	2	1.0	1.3	1.1	0.7	0.1	0.0
chart-26	12435	103	43.6	79.1	64.7	52.0	40.2	33.9

> A small amount of flakiness can affect the mutation score (less than 0.1 nominal flake rate inflates the score by 2%–5%), yet this effect diminishes as the flakiness degree increases. This results in an asymptotic growth with the mutation score plateauing around 4%-8%.

6.2 RQ2: Effectiveness of Program Repair

6.2.1 Deterministic mutation technique. To evaluate the impact of flakiness on PRAPR, we replicate its original experiments [11]. We retrieve the set of patches P generated by PRAPR and the set of valid patches V, as well as the set of covering tests T_v for each mutant. Then, we re-execute PRAPR and report the number of valid patches for different nominal flake rates (i.e. 0.1, 0.2, 0.3, 0.4, and 0.5). Even though the results of PRAPR execution are deterministic, the validity of a patch may vary from one execution to the other because of the injected flakiness. To alleviate any bias due to this randomness, we execute PRAPR 10 times for any flake rate and report the average number of valid patches over these 10 runs.

Table 1 shows the results. The number of valid patches is reduced by 17.75% (math-50) to 80% (time-19) when a nominal flake rate of 0.1 is introduced. When the degree of flakiness is increased to 0.5, the number of patches drops under 50% of the original number for all projects and PRAPR generates no valid patch for 7 out of the 16 projects. The initially large number of valid patches generated for math-50 (i.e. 40 patches) combined with the low number of covering tests (3) explains the low impact of flakiness on this project. On the contrary, for time-19, PRAPR initially generates a



(a) Mutation score **(b) Standard deviation** **(c) Difference**

Figure 2: Impact of flakiness on the mutation score. Figure 2a shows the $\overline{MS}$ of the complete test suites, while Figure 2b shows the standard deviations of $\overline{MS}$. Figure 2c shows the difference in $\overline{MS}$ that flakiness introduces.

low number of valid patches (2) while the number of covering tests is high (705 tests). In this case, the introduction of flakiness completely annihilates PRAPR's patch generation capabilities. Overall, we observe that if the number of covering tests is important the number of valid patches decreases faster. The decrease, however, remains relatively slow when compared to the results of ARJA (see Section 6.2.2).

Although PRAPR relies on SBFL to locate suspicious statements, the tool takes as input a set of initially failing tests. During the evaluation of the suspicious statements, any statement which is not covered by any initially failing test is discarded. Thus, the set of suspicious statements is bounded by the set of statements covered by all initially failing tests. Consequently, the maximum number of added tests that are introduced during the patch validation steps are those with an intersection (in terms of coverage) with the initial failing tests. The chances for at least one test to generate a flaky failure remain lower as the total number of executed tests remains low.

In an attempt to better analyze the benefits of knowing the real failing tests, we ran PRAPR without this prior knowledge. In this case, PRAPR runs first a fault localization procedure where it executes all tests. If flaky failures occur, they will augment the set of failing tests that PRAPR considers during repair. When running under these settings, for all the buggy versions contained in the dataset, PRAPR failed to generate any valid patches as soon as the slightest flakiness is introduced. This is because PRPAR repairs single faults and narrows its search to statements covered by all failing tests. If two failing tests have a disjoint coverage – which can happen if one such test is flaky – then PRAPR will generate no patch.

> As the degree of flakiness of the test suite increases, the effectiveness of deterministic mutation-based repair techniques decreases. The number of generated valid patches drops by 20% to 80% for a low degree of flakiness and exhibits a decrease of 50% to 100% when a high degree of flakiness is injected.

6.2.2 Genetic programming-based technique. We first run ARJA on each unmodified buggy program 10 times and analyze the number of valid patches. Then, we repeat the same experiment while introducing flakiness. We assign each of these tests with a nominal flake rate ranging from 0.00 to 0.20 – we stop there because at this point ARJA fails to generate any patch for all projects. For each project, we perform 10 runs of ARJA on each resulting flaky program.

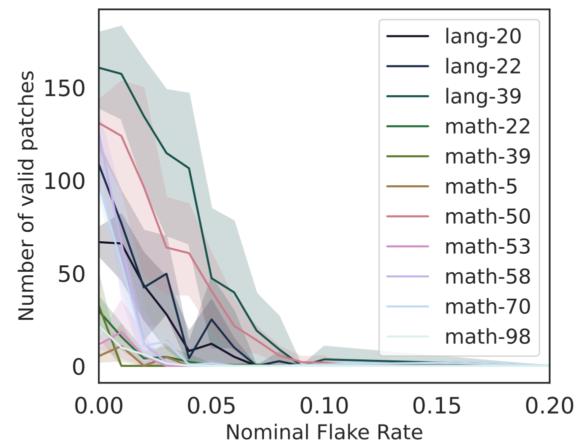


Figure 3: Sensitivity of ARJA's effectiveness to flakiness. The effect of flakiness is case-dependent and has disastrous consequences. Fewer valid patches are generated as the nominal flake rate increases. The most affected bugs are those that ARJA can hardly fix even in the absence of flakiness.

Figure 3 shows the number of valid patches generated by ARJA as the nominal flake rate increases. The results reveal that flakiness has disastrous effects on all projects and drastically reduces the number of valid patches generated – even more so as the nominal flake rate increases.

When we consider a 0.05 nominal flake rate, the average number of patches generated drops by more than 60% for all projects. Math-5, math-39, math-53, and math-98 see their score reduced by 100%, i.e. ARJA produces no valid patch for these projects. In math-22 and math-58, the number of generated patches drops from an average of 29.6 and 125 generated patches, respectively, to less than 1 patch. For math-70, this average number drops from 97.8 to 1.3. ARJA still produces valid patches for lang-20 (-82%), lang-22 (-77%) and

lang-39 (-71%). Math-50 exhibits the smallest decrease – from 130.8 to 40.3 valid patches, that is, -69%. Overall, while the impact of flakiness varies a lot from one buggy program to the other, the most negative scenarios tend to occur in programs for which ARJA could hardly generate a valid patch already in the non-flaky variants.

> The decrease in the effectiveness of genetic programming-based program repair due to flaky failures is dramatic for all bugs in our dataset. With a nominal flake rate of 0.1, the number of generated patches drops to zero for all projects.

6.3 RQ3: Targeted Fault Localization in Program Repair

As a first step to investigate a mitigation strategy for the corruption of ARJA's search space due to flakiness, we retrieve additional information from our RQ2 experiments. The failing and the passing tests that cover one or more suspicious statements identified by SBFL determine the set of statements considered to produce patches. Their number is, thus, an indication of both the size of the search space and the risk of discarding a valid patch due to flakiness.

Figure 4 shows the number of those tests for all projects and nominal flake rates, averaged over 10 runs. As the nominal flake rate increases, the number of failing tests increases almost linearly (Figure 4a). Interestingly, as soon as flakiness is introduced, the number of executed passing tests may not only increase but also *decrease* (Figure 4b).

We explain this potential decrease by the fact that Ochiai – the statement suspiciousness formula used by ARJA – depends on the number of the failing tests that cover the statement and the total number of failing tests. Hence, a flaky failure can decrease the suspiciousness of a statement if the corresponding flaky test does not cover the statement. If the suspiciousness score of the statement goes below the predefined threshold used by ARJA, then the statement is ignored during patch generation. This can, then, result in a reduction in the number of executed passing tests. We investigate the impact of flakiness on the suspiciousness score in more depth in RQ4.

We notice a general trend where the number of executed tests increases with the nominal flake rate. This is especially true, e.g., for math-22 which sees the number of tests executed explode after a nominal flake rate of 0.14. These observations shed some light on the results of RQ2: We suspect that the number of patches generated dramatically decreases because of the larger number of (flaky) tests executed.

We, therefore, pursue our investigation by studying the practical benefits of making SBFL target the real failing test cases. We conduct controlled experiments where we compare the number of valid patches produced by ARJA in (1) the previous flaky case where SBFL is applied as is and (2) a new case where no flakiness is injected during fault localization, thereby leaving the search space untouched. Doing so allows us to discard any suspicious statements and tests that are artificially added (due to flakiness occurring during SBFL), under the same nominal flake rate. As before, we run ARJA 10 times on each variant of each buggy program. Because the patch search space is not compromised, we expect to observe improvements in the number of valid patches.

Figure 5 shows the number of valid patches in the new case (`Targeted`) and in the previous case (`Non-Targeted`), with a nominal flake rate of 0.05. We observe a clear improvement in the targeted case. ARJA is even able to generate valid patches for the four programs (math-39, math-5, math-53, and math-98) it could not repair in the non-targeted case. For the remaining programs, the median number of valid patches is multiplied by a factor ranging from 2 (math-50) to 600 (math-58). A Wilcoxon signed-rank test with $\alpha = 0.05$ reveals that the differences are statistically significant (p-value $< 10^{-2}$).

These results show the importance of identifying the failing test cases on which to apply fault localization, in order to avoid corrupting both the patch search space and the validation process.

> The fault localization step of program repair is particularly vulnerable to flakiness. An efficient strategy to mitigate the effect of flakiness on APR is, therefore, to make sure that the repair targets the real failing tests. Applying this strategy has allowed the generation of patches for four more programs and yields up to 600 times more valid patches.

6.4 RQ4: Suspicious Statement Selection

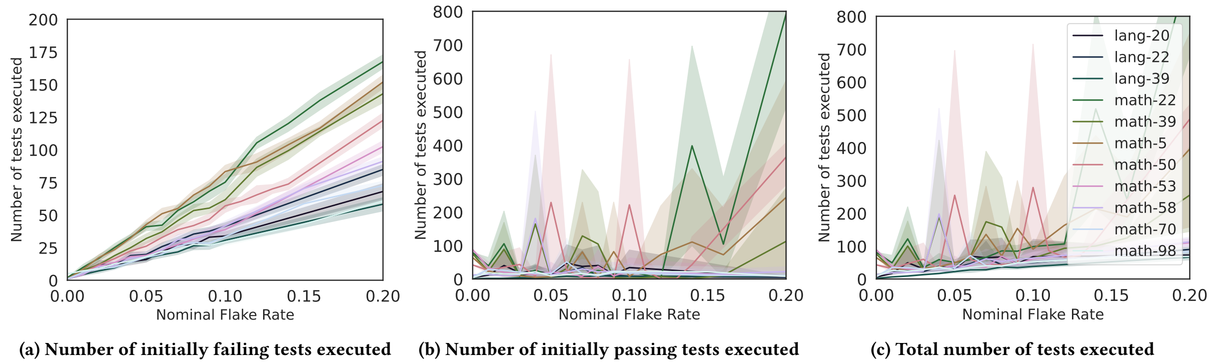
To evaluate how the alterations in the suspiciousness scores impact the set of statements selected by repair methods, we record, for each unmodified buggy program, the statements retained by ARJA after filtering. As recommended in the original paper [43], all statements with an Ochiai score below 0.1 are discarded. Taking the set of selected/discarded statements as the ground truth, we compute their counterparts in flaky variants of the programs. We, then, define the resilience of the Ochiai score against flakiness as its capability to preserve the original set of suspicious statements.

We measure this resilience using the standard metrics of accuracy, precision, and recall. Accuracy indicates the percentage of statements that remain in their class (selected or discarded), a coarse-grained view of how much the sets of statements are altered. Precision measures the percentage of selected statements (in the flaky case) that indeed had to be selected (were selected in the non-flaky case). Thus, the lower the precision, the more the patch search space is increased with patches that do not target the real buggy statements. Recall measures the percentage of real suspicious statements that remain selected in the flaky case. A lower recall means a higher risk of discarding the real buggy statements.

We compute the accuracy, precision and recall for different flaky failure rates, ranging from 0.00 (non-flaky case) to 0.50, in steps of 0.01. For each rate, we repeat the experiment 10 times.

Figure 6 shows the results for different nominal flake rates. In Figure 6a, we observe that the accuracy tends to slightly decrease with a higher rate of flaky failures but remains high for all projects (over 97% for a nominal flake rate of 0.50). Indeed, most statements are not covered by any failing test in the non-flaky program and remain so in the flaky cases.

Flakiness has, however, a drastic impact on precision and recall, with an amplitude that is project-dependent. As soon as a small



(a) Number of initially failing tests executed **(b) Number of initially passing tests executed** **(c) Total number of tests executed**

Figure 4: Number of tests (failing, passing, and total) covering one or more suspicious statements and executed by ARJA, for a nominal flake rate ranging from 0.0 to 0.2 and across 10 runs for each rate. Test flakiness creates discrepancies in the test results that are executed against candidate patches. This has a double effect: waste of computation resources and a higher risk of altering important test signals.

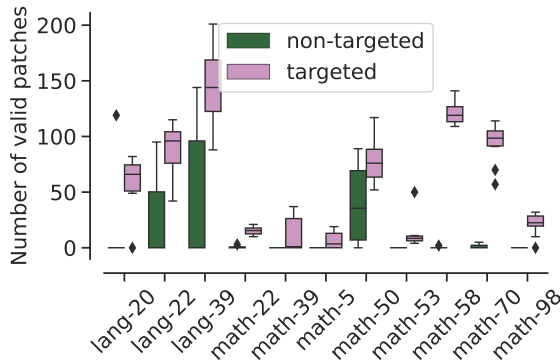


Figure 5: Number of generated valid patches obtained when no flaky failures occur during the fault localization step of ARJA (`Targeted`) and flaky failures can arise at any step (`Non-targeted`) with a nominal flake rate of 0.05.

degree of flakiness (0.01) is introduced, some projects see their precision and recall drop dramatically (e.g. math-22 which sees its precision drop to 20.82% and its recall to 35.12%; math-39 falls to a precision of 32.56% and to a catastrophic recall 6.22%; math-50 sees both precision and recall fall to about 25%).

When the nominal flake rate is set to 0.05, all projects see their precision drop by 60% or more while the recall remains more case-specific. The Ochiai score of the *Lang* projects better resist flakiness and manages to keep acceptable precision and recall at the lower flakiness rates. The precision exhibits low values (less than 50%) around at a nominal flake rate of 0.09. At 0.25, the values of recall for lang-20 and lang-22 suffer a significant drop and it is only around 0.28 that the recall of lang-39 drastically drops. Finally, on math-70 and math-98, the Ochiai-based selection offers acceptable recall until around a nominal flake rate of 0.10, where the recall drops below 50%.

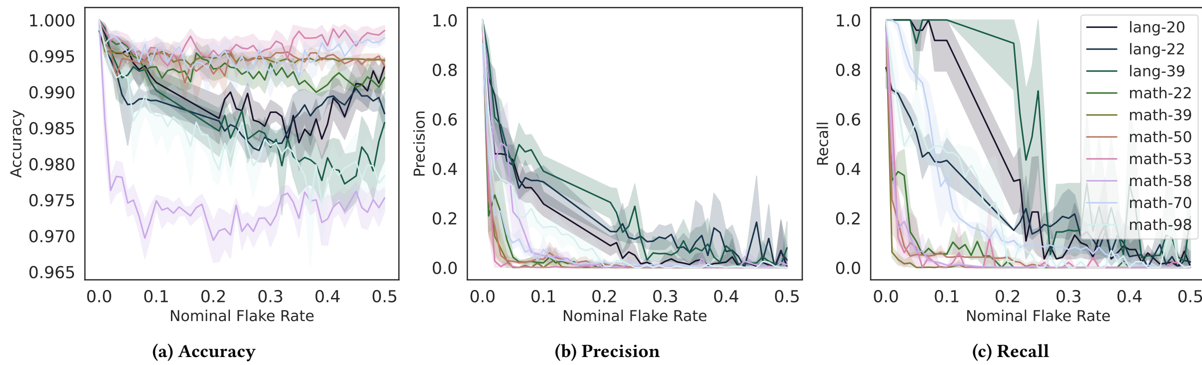
We conclude that the slightest degrees of flakiness (i.e., 0.01) can reduce the precision and recall of the threshold-based suspicious statement selection by up to 80%. This shows that the adopted threshold is yet another factor contributing to ARJA's loss of effectiveness. As shown by our results, the potential benefits of this threshold (reducing the number of tests to execute) must be balanced with the risk of executing flakiness, which can dramatically reduce the performance of program repair. Lowering the threshold may help, but still necessitates a clear *a priori* knowledge of the particular flaky failure rate.

> The smallest degree of flakiness is sufficient to disrupt threshold-based suspicious statement selection. We found that both precision and recall can drop by more than 80%. Without user feedback, SBFL cannot target real failing tests, so the use of the threshold should be avoided.

7 THREATS TO VALIDITY

The most important threat to validity regards the realism of the injected flakiness and the lack of validation of FlakiMe against the actual behaviour exposed by real flaky tests. The results of performing mutation testing or program repair with real flaky tests may, therefore, differ from the results observed with FlakiMe. While the experimental analysis that FlakiMe enables is important to stress-test the research techniques and study their limits, empirical analysis involving real flaky tests remains essential to validate these research techniques in practice. It is to be noted that FlakiMe opens the possibility for mitigating this threat and conducting empirical studies through its calibration of its behavior to a particular context. This can be done by incorporating a context-specific flakiness prediction model, e.g., to simulate particular causes or instances of flakiness. Nevertheless, the primary contribution of this paper is the experimental analysis of the impact of flakiness as enabled by FlakiMe. Our results corroborate and expand the findings of other studies conducted on real-world flakiness [36, 37, 40].

Threats to construct validity are related to how we inject flaky failures. Given that FlakiMe alters the execution of tests (and not the source code), we cannot systematically control the execution flow of the program under test. One of the major consequences of this limitation regards flakes that instead of creating flaky failures, cause it to pass. Because in its current implementation FlakiMe introduces flaky failures, it cannot simulate this behaviour. Previous studies



(a) Accuracy

(b) Precision

(c) Recall

Figure 6: Performance of suspicious statement selection as the flaky failure rate increases.

have shown that this phenomenon can occur in reality [37] and decrease the mutation score, although to a limited extent. Moreover, the mutation testing tool we use (PIT) computes test coverage once and for all on the basis of the original program and does not execute the tests not covering the mutated statements. Thus, later changes of coverage due to flakiness remain undetected by PIT and do not affect the mutation score. Nevertheless, FlakiMe can support a plethora of scenarios, such as the above ones, providing experimental control on flakiness. We, therefore, expect that future work will further alleviate such threats by considering further test suites, projects characteristics, and flakiness *injection methods*.

A threat to the internal validity is due to the flakiness prediction model that we used to determine the probability of tests to be flaky. This model relies on the vocabulary of tests and do not include any dynamic information such as coverage. Although, one could use better predictors, based on the code under test, these are unlikely to affect our results given the study of Haben *et al.* [13] that found negligible improvement on the use of the code under test.

Finally, a threat to the external validity, which hinders the ability to generalize our results, regards the selection of the projects in this study. We mitigate this threat by selecting projects from the well-known benchmark Defects4J and ensure that the flakiness probability distribution present in the test subjects (Figure 1) presents differences. Furthermore, all the projects taken in our analysis are written in Java, but we do not have any evidence that different languages would yield significantly different results.

Ultimately, although laboratory-controlled conditions inevitably differ from reality to some extent, they can still lead to interesting insights that remain applicable in the real world, such as the mitigation strategies that our study has allowed us to reveal. We believe that one of the main strengths of FlakiMe remains its flexibility to integrate different "flakiness models" – i.e. the set of methods and conditions that determine how FlakiMe injects flakiness in programs and tests. We entrust future users of FlakiMe with the task of designing appropriate models for the specific flakiness phenomena that these researchers investigate – and do so with the awareness that any model they build comes with inevitable threats to validity and should, therefore, be used under specific and validated working assumptions. To ease this endeavor, additionally to the FlakiMe Maven plugin, we provide a full replication package.[7]

[7] https://github.com/serval-uni-lu/flakime-replication-package

8 CONCLUSION

We presented a test flakiness platform, FlakiMe, that allows experimenting with laboratory-controlled test flakiness. FlakiMe is customizable and can simulate a wide range of flakiness-related conditions and scenarios. To demonstrate the utility of FlakiMe we performed laboratory-controlled experiments to assess the impact of test flakiness on mutation testing and automated program repair.

Interestingly, we showed that putting flakiness under laboratory control adds a new dimension to software testing studies, which is the simulation of a world where some tests exhibit non-deterministic behaviour and are considered as *potentially* flaky. Such a world allows establishing a better understanding of the effects of flakiness and paves the way for developing robust (against flakiness) test techniques.

For instance, we demonstrated that mutation testing, a popular test assessment metric, is impacted by flaky tests, *i.e.*, mutation score is inflated by approximately 2%-8% depending on the degree of flakiness. This effect is however small, as the introduced noise is similar among all cases making the metric relatively stable.

In program repair, our results showed that the fault localization step is particularly sensitive to test flakiness. Such sensitivity can have disastrous effects on patch generation. Thus, to make program repair techniques robust to flaky tests, one should revisit the key decisions and assumptions made during fault localization.

For example, in a scenario where some 'real' failing tests are specified as inputs, a tailored fault localisation procedure that considers only these tests helps to prevent the corruption of the patch search space as well as useless runs of the candidate patches with flaky test cases. With FlakiMe as a framework to conduct controlled and fine-grained experiments, researchers can further analyze mitigation techniques to improve the resilience of software testing techniques under flakiness.

ACKNOWLEDGEMENT

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C20/IS/14761415/TestFlakes. Mark Harman's scientific work is part supported by European Research Council (ERC), Advanced Fellowship grant number 741278; Evolutionary Program Improvement (EPIC) which is run out of University College London, where he is part time professor. He is a full time Research Scientist at Meta.

REFERENCES

[1] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. 2019. Empirical Analysis of Factors and their Effect on Test Flakiness - Practitioners' Perceptions. *CoRR* abs/1906.00673 (2019). arXiv:1906.00673

[2] Nadia Alshahwan, Andrea Ciancone, Mark Harman, Yue Jia, Ke Mao, Alexandru Marginean, Alexander Mols, Hila Peleg, Federica Sarro, and Ilya Zorin. 2019. Some Challenges for Software Testing Research (Invited Talk Paper). In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/3293882.3338991

[3] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140

[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. ACM, 433–444. https://doi.org/10.1145/3180155.3180164

[5] B. H. P. Camara, M. A. G. Silva, A. T. Endo, and S. R. Vergilio. 2021. What is the Vocabulary of Flaky Tests? An Extended Replication. In *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension*. IEEE/ACM, 11.

[6] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[7] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3338906.3338911

[8] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 211–224. https://doi.org/10.1145/3395363.3397366

[9] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. 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*. ACM, 830–840. https://doi.org/10.1145/3338906.3338945 arXiv:1907.01466

[10] Zebao Gao and Atif M. Memon. 2015. Which of My Failures are Real? Using Relevance Ranking to Raise True Failures to the Top. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 62–69. https://doi.org/10.1109/ASEW.2015.7

[11] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[12] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 11. arXiv:2101.09077

[13] Guillaume Haben, Sarra Habchi, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2021. A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests. In *Proceedings of the 18th International Conference on Mining Software Repositories*. ACM, 11.

[14] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis (keynote paper). In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation*. 1–23.

[15] Facebook Inc. 2019. Facebook Testing and Verification request for proposals.

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[17] Tariq M. King, Dionny Santiago, Justin Phillips, and Peter J. Clarke. 2018. Towards a Bayesian Network Model for Predicting Flaky Automated Tests. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*. IEEE, 100–107. https://doi.org/10.1109/QRS-C.2018.00031

[18] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM, 110–119. https://doi.org/10.1145/3377813.3381370

[19] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 821–830. https://doi.org/10.1145/3106237.3106288

[20] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing, China, 101–111. https://doi.org/10.1145/3293882.3330570

[21] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, 1471–1482. https://doi.org/10.1145/3377811.3381749

[22] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 312–322. https://doi.org/10.1109/ICST.2019.00038

[23] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-test-aware regression testing techniques. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 298–311. https://doi.org/10.1145/3395363.3397364

[24] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. 2021. When life gives you oranges: detecting and diagnosing intermittent job failures at Mozilla. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 1381–1392. https://doi.org/10.1145/3468264.3473931

[25] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[26] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing Transition-Based Test Selection Algorithms at Google. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, 101–110. https://doi.org/10.1109/ICSE-SEIP.2019.00019

[27] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In 22nd *International Symposium on Foundations of Software Engineering (FSE 2014)*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne Storey (Eds.). ACM, Hong Kong, China, 643–653.

[28] Jean Malm, Adnan Causevic, Björn Lisper, and Sigrid Eldh. 2020. Automated Analysis of Flakiness-mitigating Delays. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. ACM, 81–84. https://doi.org/10.1145/3387903.3389320

[29] Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In *Proceedings of the 35th International Conference on Software Engineering*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, San Francisco, CA, USA, 1479–1480.

[30] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In 39th *International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, Buenos Aires, Argentina, 233–242.

[31] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*, Atif M. Memon (Ed.). Advances in Computers, Vol. 112. Elsevier, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[32] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2020. Flake It 'Till You Make It. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ACM, 11–12. https://doi.org/10.1145/3387940.3392177

[33] Samad Paydar and Aidin Azamnouri. 2019. An Experimental Study on Flakiness and Fragility of Randoop Regression Test Suites. In *Fundamentals of Software Engineering*, Hossein Hojjat and Mieke Massink (Eds.). Springer International Publishing, 111–126. https://doi.org/10.1007/978-3-030-31517-7_8

[34] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo D'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the Vocabulary of Flaky Tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, 492–502. https://doi.org/10.1145/3379597.3387482

[35] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn T Stolee. 2019. Wait Wait . No , Tell Me . Analyzing Selenium Configuration Effects on Test Flakiness .. In *Proceedings of the IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. IEEE, Montreal, Canada, 2–8. https://doi.org/10.1109/AST.2019.000-1

[36] Yihao Qin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawende F. Bissyande. 2021. On the Impact of Flaky Tests in Automated Program Repair. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 295–306. https://doi.org/10.1109/SANER50967.2021.00035

[37] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT*

International Symposium on Software Testing and Analysis. ACM, 112–122. https://doi.org/10.1145/3293882.3330568

[38] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. 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 (ESEC/FSE 2019)*. Association for Computing Machinery, 545—-555. https://doi.org/10.1145/3338906.3338925

[39] Denini Silva, Leopoldo Teixeira, and Marcelo D'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE, 301–311. https://doi.org/10.1109/ICSME46990.2020.00037

[40] Bela Vancsics, Tamas Gergely, and Arpad Beszedes. 2020. Simulating the Effect of Test Flakiness on Fault Localization Effectiveness. In *Proceedings of the IEEE Workshop on Validation, Analysis and Evolution of Software Tests*. IEEE, 28–35. https://doi.org/10.1109/VST50071.2020.9051636

[41] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. 2021. Know You Neighbor: Fast Static Prediction of Test Flakiness. *IEEE Access* 9 (2021), 76119–76134. https://doi.org/10.1109/ACCESS.2021.3082424

[42] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[43] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648