What Made This Test Flake? Pinpointing Classes Responsible for Test Flakiness

Sarra Habchi
Ubisoft
sarra.habchi@ubisoft.com

Guillaume Haben
University of Luxembourg
guillaume.haben@uni.lu

Jeongju Sohn
University of Luxembourg
jeongju.sohn@uni.lu

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

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

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

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

***Abstract*—Flaky tests are defined as tests that manifest non-deterministic behaviour by passing and failing intermittently for the same version of the code. These tests cripple continuous integration with false alerts that waste developers' time and break their trust in regression testing. To mitigate the effects of flakiness, both researchers and industrial experts proposed strategies and tools to detect and isolate flaky tests. However, flaky tests are rarely fixed as developers struggle to localise and understand their causes. Additionally, developers working with large codebases often need to know the sources of non-determinism to preserve code quality, *i.e.,* avoid introducing technical debt linked with non-deterministic behaviour, and to avoid introducing new flaky tests. To aid with these tasks, we propose re-targeting Fault Localisation techniques to the flaky component localisation problem, *i.e.,* pinpointing program classes that cause the non-deterministic behaviour of flaky tests. In particular, we employ Spectrum-Based Fault Localisation (SBFL), a coverage-based fault localisation technique commonly adopted for its simplicity and effectiveness. We also utilise other data sources, such as change history and static code metrics, to further improve the localisation. Our results show that augmenting SBFL with change and code metrics ranks flaky classes in the top-1 and top-5 suggestions, in 26% and 47% of the cases. Overall, we successfully reduced the average number of classes inspected to locate the first flaky class to 19% of the total number of classes covered by flaky tests. Our results also show that localisation methods are effective in major flakiness categories, such as concurrency and asynchronous waits, indicating their general ability to identify flaky components.**

I. Introduction

Regression testing is a key component of continuous integration (CI) that checks whether code changes integrate well in the codebase without breaking any existing functionality. To this end, it is assumed that failing tests indicate the presence of faults, introduced by the latest changes. However, some tests break this assumption by failing for reasons other than faults, as for instance, they exhibit non-deterministic behaviour, thereby sending confusing signals to developers. Such tests are usually called *flaky tests*.

Academic and industrial reports have emphasised the adverse effects of test flakiness in software development. Specifically, Google reported that 16% of their tests manifested some level of flakiness, while more than 90% of their test transitions, either to failing or passing, were due to flakiness [1]. As the *de facto* approach for detecting flaky tests is to rerun them [2], [3], detecting large numbers of flaky tests can be time- and resource-consuming. Indeed, Google reports that between 2 to 16% of their CI resources are dedicated in rerunning flaky tests [4]. It is noted that other companies, like Microsoft [5], Spotify [6] and Mozilla [7], also report similar issues when dealing with test flakiness.

Perhaps more importantly, test flakiness affects team productivity and software quality [2]. This is because flaky failures interrupt the CI and make developers waste time in investigating false issues [8], [9], [1], [2]. Additionally, the accrual of flaky tests breaks the trust in regression testing, leading developers to disregard legitimate failure signals believing them to be false [2], [3]. This situation often results in faults slipping into production systems [7]. Moreover, code quality is often linked with the level of flakiness incurred [2] and thus, developers need to know where it comes from and understand the causes of flakiness to avoid introducing and spreading it.

Given the adverse effects of test flakiness, engineers and researchers aim at developing detection techniques that can predict whether a test is potentially flaky. These approaches rely on a number of runs and re-runs, such as iDFlakies [10] and Shaker [11], coverage analysis like DeFlaker [12], or static and dynamic test features [13], [14], [15], [16], [17], [18], [19]. Evaluated on open-source projects, these approaches showed promising detection accuracy and considerably decreased the amount of time and resources needed to detect flaky tests.

Although flakiness detection methods are important, alone, they cannot reduce the prevalence of test flakiness. This is because on the one hand there are only partial approaches to the problem, such as iFixFlakies [20] and Flex [21] that are only applicable to specific cases, and the inherent difficulties in isolating/controlling the flakiness causes on the other. For instance, iFixFlakies [20] fixes order-dependent tests by identifying helper statements in other tests, whereas Flex [21] identifies assertion bounds that minimise flakiness stemming from algorithmic randomness. At the same time, many prevalent categories of flakiness, *e.g.,* Asynchronous Waits and

Concurrency [22], [23], [24], [9], remain unaddressed by fixing approaches. This is mainly due to the difficulty of identifying and controlling the cause of flakiness [9].

Flakiness root cause localisation is both important and difficult. It is important since it allows developers to understand the sources of flakiness, hence enabling better control of non-determinism. It is also difficult because of the difficulty to reproduce failures, the diversity in potential issues, *e.g.,* time and network, and the large scope of potential culprits, *e.g.,* the tests, the code under test (CUT), and the infrastructure [22]. Consequently, practitioners struggle to identify the causes of non-determinism in their codebases that trigger flakiness and consider this step as the main challenge in automating flakiness mitigation strategies [9].

In this paper, we address this challenge by re-targeting Fault Localisation (FL) techniques in order to help identify components (program classes in particular) that are responsible for the non-deterministic behaviour of flaky tests. For the sake of simplicity, we refer to these classes as *flaky classes*. Such techniques can be useful to support the analysis of codebases and of flaky tests. Thus, given a failure, either known as flaky or unknown, engineers can rely on localisation methods to investigate the specific scenario (condition) that causes the test transition. Additionally, flakiness localisation techniques can help with code comprehension and make engineers aware of code areas linked with flaky behaviour, assisting them in both development and testing tasks.

In view of this, we investigate the appropriateness of a variety of fault localisation methods, such as Spectrum-Based Fault Localisation (SBFL), change history metrics, and static code metrics in identifying flaky classes. Our study aims to answer the following four research questions:

- **RQ1:** Are SBFL-based approaches effective in identifying flaky classes?
- **RQ2:** How do code and change metrics contribute to the identification of flaky classes?
- **RQ3:** How can ensemble learning improve the identification of flaky classes?
- **RQ4:** How does an SBFL-based approach perform for different flakiness categories?

To answer these questions, we analyse five Open Source projects where test flakiness has been fixed during the project evolution. Our analysis shows that:

- An ensemble of models based on SBFL, change, and size metrics, yields the best results, with 61% of flaky classes in the top 10 and 26% of them at the top. This method also reduces the average effort wasted by developers to 19% of the effort spent when inspecting all classes covered by the flaky test.
- The ensemble method is effective for major flakiness categories. Concurrency and Asynchronous Waits are identified effectively, with 38% and 30% of their flaky classes ranked at the top, respectively.

To facilitate the reproducibility of this study, we provide all used scripts, the set of collected flaky classes, and detailed results in a comprehensive package[1].

TABLE I: Collected Data. *ffc:* number of flakiness-fixing commits. *all:* number of commits in the project.

Proj.	#Commits		#Tests		#Classes	
	ffc	all	min - max	avg	min – max	avg
Hbase	8	18,990	138 - 2,089	1,113	734 – 1366	1053.4
Ignite	14	27,903	15 - 1,018	174	72 – 1767	1262.3
Pulsar	10	8,516	194 - 1,326	626	171 – 422	259.7
Alluxio	3	32,560	315 - 694	473	131 – 817	360.3
Neo4j	3	71,824	21 - 5,782	2,139	40 – 1663	581.3
Total	38		15 - 5,782	905	40 – 1767	820.2

II. Data Collection

The objective of our study is to assess the effectiveness of FL techniques in identifying flaky classes. To achieve this, we need a set of flaky tests for which the responsible classes are already known. For this, we rely on flakiness-fixing commits as they provide information about classes that were modified as part of the fix. Our assumption is that such classes are, at least, part of the root cause. To collect flaky classes, we followed a four-step process.

a) Search: This step aims to identify Java projects containing the highest number of flakiness-fixing commits. For this, we relied on two sets of projects to consider. We built the first set by using the SEART GitHub Search Engine [25]. Out of the 81,180 available Java projects, we selected the top 200 projects for each of those criteria: number of commits, contributors, stars, releases, issues, and files. This sorting was made with the aim of finding the bigger and more complex projects, thus maximising our chance to find flakiness-fixing commits. Keeping only unique projects in those sets, we ended up with a first list of 902 projects. As a second set, we use the 187 projects available in the iDFlakies dataset [10]. For each of the 1,089 projects, 902 from the first and 187 from the second set, we query the GitHub API looking for commits with messages containing the keyword *flaky*. This led to the identification of 16,501 commits. We look further into whether these commits are truly suitable for our purpose through the following processes.

b) Inspection: The objective of this step is to filter commits that do not provide a clear indication about the flaky class. Hence, we look for flakiness-fixing commits containing any of the following keywords: *fix, repair, solve, correct, patch, prevent*. Then, we analyse each commit and keep the ones that:
- The fix affects the code under test (not only the test itself);
- The changes are atomic enough (*i.e.,* containing only relevant changes) allowing us to discern the flaky class(es).

This led to the selection of 85 commits from five projects. We further discarded 22 commits for which the flaky tests or commit were not retrievable (*e.g.,* rejected pull request), leaving 63 commits in the end.

c) Test execution: This step aims to select commits that are usable in our evaluation. Our first question inspects the effectiveness of SBFL, a technique that requires a coverage

[1]https://github.com/serval-uni-lu/sherlock.replication

matrix indicating the classes covered by each test. Hence, for a commit to be usable in our analysis, its test suite should be runnable allowing us to extract the coverage matrix. To ensure this, we used GZOLTAR[2], a Maven plugin that allows collecting coverage information for each commit. For 11 commits, we were unable to run GZOLTAR due to an incompatible Java version. We also found that the flakiness patches were irrelevant in 10 commits. For instance, some commits were fixing modules in other programming languages or modifying non-source code files. Lastly, we filtered out four additional commits since the reported flaky failures were not *flaky test failures*. Consequently, we dropped 25 commits in addition. Table I summarises the retained projects. The complete list of flakiness-fixing commits is available in our replication package.

d) Extraction: For each collected flakiness-fixing commit, we retrieve the source code, the test suite, the fixed flaky test, and the flaky class. To retrieve the flaky classes, two authors manually analysed the commit diff and message to identify them. Overall, the identification was obvious since we selected atomic commits beforehand. Hence, there were no disagreements between the authors at this step. The identified classes are considered the ground truth of our study.

III. STUDY DESIGN

A. RQ1 - Effectiveness

1) Motivation: The objective of our study is to investigate the usability of well-founded FL techniques to help in mitigating flaky tests. The literature on FL proposes a wide variety of categories such as ML-based techniques [26], [27], [28], mutation-based techniques [29], [30], and qualitative reasoning-based techniques [31]. Nonetheless, spectrum-based fault localisation remains one of the most distinguished FL categories thanks to its effectiveness and simplicity [32]. SBFL requires only the test coverage matrix to compute the likelihood for a code entity to include the root cause of an observed test failure. The main assumption of SBFL is that code entities covered by more failing tests and fewer passing tests are more suspicious than those less covered by failing tests and more by passing tests [33]. This assumption can be revised to identify the root causes of flaky tests instead of bugs. In particular, if we separate tests into two groups: *flaky* and *stable*, instead of *failing* and *passing*, we can leverage the coverage matrix to rank classes based on their correlation with flaky tests. In this case, the assumption would be that classes covered by more flaky tests and fewer stable tests have a higher chance to be responsible for test flakiness. In this RQ, we assess the effectiveness of this adaptation of SBFL in identifying flaky classes.

2) Approach: Relying on the data collected in Section II, we use the GZOLTAR plugin to run the test suites of each commit and build coverage matrices. Based on these matrices, we compute for each class the spectrum data: (e_s, e_f, n_s, n_f). In our case, for each class, e_s and e_f represent the number of stable and flaky tests executing it, respectively. On the other hand, n_s and n_f represent the number of stable and flaky tests that do not execute it, respectively. To compute classes' suspiciousness scores, we inject these spectrum data in classical SBFL formulæ. Table II summarises the four formulæ adopted in our study with the necessary adaptations for flakiness. For DStar, the notation '*' is a variable that we set to 2 based on the recommendation of Wong *et al*. [34]. With each formula, we compute the suspiciousness scores of each class and then rank them in descending order: classes with the highest scores are ranked first.

TABLE II: SBFL formulae adapted to flakiness.

Name	Formula
Ochiai [42]	$\frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_s)}}$
Barinel [43]	$1-\frac{e_s}{e_s+e_f}$
Tarantula [44], [45]	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_s}{e_s+n_s}}$
DStar [34]	$\frac{e_f^*}{e_s*n_f}$

Recently, it has been theoretically proven that no SBFL formula can outperform all others [35]. In addition, Xuan and Monperrus proposed a new approach that learns to combine multiple SBFL formulæ [36]. Their approach, called Multric, successfully outperformed all the input formulæ, opening a trend to use multiple formulæ to overcome the limitation of using a single SBFL formula [37], [38], [27]. Following this trend, we used Genetic Programming to evolve a new formula that combines all four SBFL formulæ.

Genetic Programming (GP) evolves a solution (*i.e.,* a program) for a given problem under the guidance of a (fitness) function. GP can also generate non-linear models and learn a model flexibly from input instead of defining a fixed formula. Hence, GP was employed to generate risk evaluation formulæ for fault localisation [39], [40]. For the same reasons, we employ GP to evolve a model (*i.e.,* a formula) for the flaky class identification problem. We configure the GP to have a population of 40 individuals and to stop and return the best model found so far after 100 generations. Each individual in the population denotes a single candidate formula and is generated using (i) six arithmetic operators (subtraction, addition, multiplication, division, square root, and negation) and (ii) the features that GP takes as input. We define our fitness function as the average ranking of flaky classes. To make most of the data and avoid overfitting, we use ten-fold cross-validation, using one fold for test and the others for training. We also normalise all input data between 0 and 1 using min-max normalisation. Finally, to compensate for the inherently stochastic nature of GP, we run GP 30 times with different random seeds and report the results of a model with the median fitness. We used DEAP v.1.3.1 [41].

B. RQ2 - Code and change metrics

1) Motivation: The objective of this question is to explore the benefits of augmenting the SBFL technique with additional signals from the software. Recent studies showed that the

[2]https://github.com/GZoltar/gzoltar/blob/master/com.gzoltar.ant/

performances of SBFL can be improved by incorporating signals from code and change metrics. More specifically, Sohn and Yoo [40] showed that combining SBFL with code and change metrics widely adopted in the fault prediction community [46], such as age, change frequency (*i.e.,* churn), and size, can significantly improve the approach's performances. The assumption is that code entities with higher complexity and change frequency are more likely to be faulty. Several studies suggested that the test size and complexity can also be an indicator of flakiness [15], [47], [18]. However, it is unclear if such metrics correlate also with classes that are responsible for test flakiness. Therefore, in this RQ, we assess the benefits of these metrics in spotting flaky classes. Besides these metrics, we investigate the effects of metrics that are specific to the nature of flaky tests. Multiple empirical studies analysed the root causes of flakiness and showed that the main categories are: Async Waits, Concurrency, Order-dependency, Network, Time, I/O operations, Unordered collections and Randomness [24], [48], [49], [22]. We derived a list of static metrics that describe each of these categories in Java projects. We exclude order-dependency because order-dependent tests generally stem from tests themselves instead of the CUT, thus, they are not concerned by our approach. In the following, we describe our approach for (i) calculating these metrics and (ii) defining a FL formulae based on them.

2) Approach:

a) Metric collection: Table III summarises the full list of metrics used in our study. To compute these metrics, we first retrieve the source code of the project at the commit of interest (*i.e.,* the parent commit of the flakiness-fixing commit identified by the data collection step). Then, for calculating flakiness-specific metrics, we use Spoon [50]. Spoon is a framework for Java-based program analysis and transformation that allows us to build an abstract syntax tree and a call graph. Using the graph and tree, we extract classes and their metrics (*e.g.,* #COPS and #ROPS). For size metrics, we also use these code analysis results from Spoon (*e.g.,* DOI). As for change metrics, we analyse the change history and extract the following information: the date of each commit, files modified and renamed by each commit, and authors of individual commits. Using this information, we compute the three change metrics: Unique Changes, Age, and Developers.

b) Ranking model: Similarly to RQ1, we use GP in order to generate models that combine our metrics with suspiciousness scores generated by SBFL formulæ. In particular, for each type of metrics (*i.e.,* flakiness, size, and change), we evolve a model that takes as input its metrics with SBFL scores and outputs a ranking for each candidate class. Afterwards, we compare the performances of these models to infer the contribution of each type of metrics.

C. RQ3 - Ensemble method

1) Motivation: This question explores the potential for improvement by exploiting all the formulæ generated using GP while at the same time making the most of the resources spent on model generation. For this aim, we use voting as our ensemble learning method. We opted for voting since it does not require an additional cost for model generation and its effectiveness has already been demonstrated by previous fault localisation studies [51], [52]

TABLE III: Code and change metrics used to augment SBFL.

	Metric	Definition
Flakiness	#TOPS	Number of time operations performed by the class.
	#ROPS	Number of calls to the `random()` method in the class.
	#IOPS	Number of input/output operations performed by the class.
	#UOPS	Number of operations performed on unordered collections by the class.
	#AOPS	Number of asynchronous waits in the class.
	#COPS	Number of concurrent calls in the class.
	#NOPS	Number of network calls in the class.
Change	Changes	Number of unique changes made on the class.
	Age	Time interval to the last changes made on the class.
	Developers	Number of developers contributing to the class.
Size	LOC	The number of lines of code.
	CC	Cyclomatic complexity.
	DOI	Depth of inheritance.

2) Approach: Voting between models is performed in two phases: candidate selection and voting. During the candidate selection phase, all the participating models compute their own suspciousness scores for the candidates. A candidate, in our case, is an individual class of the CUT. Individual models compute their own suspiciousness scores for the candidates and select those placed within the top N as their candidates to vote. In the voting phase, each model votes for its own top N candidates. If M number of models participate in the voting, we can have the maximum $N \times M$ number of voted candidates in total. The votes from the models are then aggregated, and the voted candidates are reordered from the most voted to the least voted.

Previous studies on voting-based FL showed that varying the number of votes that each candidate receives based on its actual rank in individual models can improve the localisation performance even further [51], [52]. Hence, rather than assigning the same number of votes to each candidate, we allow individual models to cast a different number of votes for each candidate based on its location in the ranking. For instance, a candidate ranked at the top will obtain a complete one vote, whereas a candidate ranked in the third place will get $\frac{1}{3}$ vote. As mentioned in III-F, candidates can be tied with other candidates since their ranks are computed from ordinal scores. When a candidate fails to be in the top N due to being tied with others, we allow every tied candidate (c) to receive the following number of votes: $votes = \frac{1}{rank_{best}(c) \times n_{tied}(c)}$ votes. Here $rank_{best}$ denotes the best (highest) rank a tied candidate can have, and n_{tied} is the total number of tied candidates, including itself. The equation below summarises the number of votes a candidate (c) can obtain. $rank(c)$ is the rank of the candidate c.

$$\begin{cases} \frac{1}{rank(c)} & \text{if } rank(c) \leq N \\ \frac{1}{rank_{best}(c) \times n_{tied}(c)} & \text{if } rank_{best}(c) \leq N \\ 0 & \text{otherwise} \end{cases}$$

D. RQ4 - Flakiness categories

1) Motivation: The literature on flaky tests reports different categories of flakiness [24], [48], [49], [22]. These categories can manifest differently both in the test and CUT and as a result the identification of flaky classes can also be affected by such differences. That is, a technique might identify decently the classes responsible for non-deterministic network operation, but struggles in pinpointing classes causing race conditions. This RQ aims to investigate the performances of an SBFL-based approach among distinct flakiness categories.

2) Approach: Many studies manually analysed flakiness-fixing commits to categorise them [24], [53] based on their commit message and code changes. In our study, we followed a similar process where two authors manually analysed the commits separately to assign them to one of the categories derived by Luo *et al*. [24]. As our manual analysis does not intend to build a new taxonomy or identify new categories, it is reasonable to adopt an existing taxonomy as reference. The two authors had a disagreement over one commit, where one author only suggested one category whereas the other suggested two categories. After discussion, the authors decided to keep two categories to avoid discarding relevant information. The results of this analysis are available in our replication package. After labelling the flakiness-fixing commits, we analyse the performance of our SBFL-based approach among different flakiness categories.

E. Evaluation metrics

For the evaluation of our approach, we use two metrics: accuracy and wasted effort. Both *acc@n* and *wef* are based on the absolute number of code entities instead of percentages. This conforms to the recommendations of Parnin and Orso [54] who suggested that absolute metrics reflect the actual amount of efforts required from developers better than percentages. The accuracy (*acc@n*) calculates the number of cases where the flaky classes were ranked in the top n. In our study, we report the *acc@n* with 1, 3, 5, and 10 as n values. In the cases of multiple flaky classes, we consider the flaky class to be among the top n, if at least one of the flaky classes is. The second metric, wasted effort (*wef*), allows us to measure the effort wasted while searching for the flaky class. It is formally defined as [36]:

$$wef = |susp(x) > susp(x*)| + |susp(x) = susp(x*)|/2 + 1/2$$

Where $susp()$ provides the suspiciousness score of the class x, $x*$ is the flaky class, and $|.|$ provides the number of elements in the set. Accordingly, *wef* measures the absolute number of classes inspected before reaching the real flaky class $x*$.

For our approach to be useful for developers, it should provide guidance beyond currently available information. When a program fails due to flaky tests, one thing that can be helpful to identify the cause is a list of classes covered by the flaky tests. Hence, in this paper, we count the total number of classes covered by flaky tests (*i.e.,* our baseline) and compare it with the number of classes inspected to locate a flaky class (*i.e.,* *wef*+1). More specifically, in addition to the two absolute metrics, we measure the relative effort defined as:

$$R_{wef} = \frac{100 \times (wef + 1)}{\text{\# of classes covered by flaky tests}},\ 0 < R_{wef} \leq 100$$

If R_{wef} is smaller than 50, we consider our approach to outperform the baseline since it saves more than the expected effort (*i.e.,* average) of the baseline.

F. Tie-breaking

Both SBFL and our evolved formulæ compute an ordinal score for each class. As a result, multiple classes can have the same score, being tied to each other. Ties are generally harmful as they force developers to inspect more classes. Among various tie-breakers introduced and adopted to handle this problem [55], we use a max tie-breaker that assigns the lowest rank (*i.e.,* the maximum) to all tied entities. We choose the max tie-breaker to avoid overinterpretation of the results.

IV. Study Results

A. RQ1 - Effectiveness

Table IV shows the localisation results of SBFL formulæ. Among the four SBFL formulæ, Dstar yields the worst results both in accuracy and wasted effort, while the other three perform similarly. Out of 38 analysed flaky classes, Dstar ranks 18 (47%) in the top 10. Ochiai, which performs the best, places 53% of flaky classes (*i.e.,* 21) within the top 10 and 16% (6) at the top. Nevertheless, regardless of which formula we use, our SBFL-based approach outperforms the baseline of inspecting classes covered by flaky tests: for all four SBFL formulæ, R_{wef} is always smaller than 50 in total, especially in its median. It is worth noting that since the total number of classes covered by flaky tests differs in each flaky commit, R_{wef} does not always concur with *wef*. For Ochiai, R_{wef} reduces to 6, meaning we only need to inspect 6% of the classes covered by flaky tests.

Table V presents the evaluation results of our GP model evolved to combine the four SBFL formulæ. As explained in Section III-A, we report only the results of the model with the median fitness among 30 models. In contrast to what we expected from combining the four SBFL formulæ using GP, we fail to observe any meaningful improvement compared to the results of Ochiai, the best of the four formulæ: the *acc@10* and the median wasted effort improve only marginally, and R_{wef} degrades.

To understand these observations, we inspect the intersection between the sets of classes ranked in the top 5 by these four SBFL formulæ. Figure 1 presents this intersection in a Venn diagram. Out of 14,16,15,15 flaky classes ranked within the top 5 by Dstar, Ochiai, Tarantula, and Barinel, 13 of them are the same flaky classes. There are two additional classes

TABLE IV: RQ1: Effectiveness of SBFL formulæ. (#) denotes the total number of flaky commits for each project. The row *Perc* contains the percentage of flaky commits whose triggering flaky classes are ranked in the top n; these values are computed only for *acc@n*.

Proj. (#)	Dstar						Ochiai						Tarantula						Barinel					
	acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})	
	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med
Hbase (8)	0	3	4	4	33.0 (17)	7 (5)	2	5	5	5	14.9 (13)	1 (4)	1	4	4	5	11.9 (12)	4 (4)	1	4	4	5	11.6 (12)	4 (4)
ignite (14)	0	2	2	2	214.7 (21)	31 (4)	0	3	3	4	212.0 (20)	20 (4)	0	3	3	4	177.1 (17)	20 (4)	0	3	3	4	175.1 (17)	20 (4)
Pulsar (10)	1	3	6	9	9.9 (21)	4 (6)	3	5	6	9	9.2 (13)	3 (6)	3	5	6	9	9.2 (13)	3 (6)	3	5	6	9	9.2 (13)	3 (6)
Alluxio (3)	0	0	0	1	60.7 (43)	72 (31)	0	0	0	1	71.0 (46)	72 (41)	0	0	0	0	92.7 (59)	73 (58)	0	0	0	0	105.3 (66)	87 (65)
Neo4j (3)	1	2	2	2	12.0 (41)	1 (18)	1	2	2	2	12.0 (41)	1 (18)	1	2	2	2	23.0 (43)	1 (18)	1	2	2	2	23.7 (43)	1 (18)
Total (38)	2	10	14	18	94.4 (24)	**11 (17)**	6	15	16	21	90.2 (21)	7 (6)	5	14	15	20	79.3 (21)	8 (7)	5	14	15	20	79.6 (21)	8 (7)
Perc (%)	5	26	37	**47**	-	-	**16**	**39**	42	**55**	-	-	13	37	39	**53**	-	-	13	37	39	**53**	-	-

TABLE V: RQ1: The effectiveness of GP evolved formulæ using Ochiai, Barinel, Tarantula, and DStar.

Project	Total	acc				wef (R_{wef})	
		@1	@3	@5	@10	mean	med
Hbase	8	1	4	5	5	13.12 (16)	2.5 (5)
Ignite	14	0	3	3	5	214.93 (21)	20.0 (4)
Pulsar	10	3	5	6	9	9.20 (23)	3.0 (9)
Alluxio	3	0	0	0	1	101.67 (65)	86.0 (83)
Neo4j	3	1	2	2	2	23.33 (43)	1.0 (18)
Total	38	5	14	16	22	94.24 (26)	6.5 (8)
Percentage (%)	100	**13**	**37**	42	**58**	-	-

that are ranked in the top 5 by all except Dstar and one extra class by only Ochiai and Dstar. Overall, the diagram demonstrates that there are large overlaps between the results of these four SBFL formulæ. Thus, we can conclude that the GP-evolved formula did not lead to substantial improvements because there was no space for improvement as all four input formulæ provided similar signals. This conclusion brings out the need for introducing external signals from other code and change metrics, which will be discussed in the following research question.

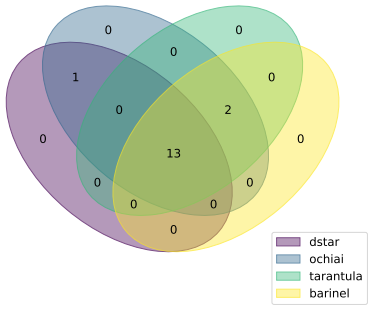

Fig. 1: Venn-diagram of flaky classes ranked in the top 5 by the four SBFL formulæ.

> Using SBFL, we were able to localise flaky classes by inspecting only 21-24% (6-7%) of classes covered by flaky tests on average (median). With Ochiai, flaky classes are ranked at the top and in the top 10 for 16% and 55% of total flaky commits.

B. RQ2 - Code and change metrics

Table VI shows the evaluation results for GP-evolved models using SBFL scores with change and code metrics. The table shows that the addition of signals from change and size metrics leads to an improvement in the identification of flaky classes. In particular, by adding change metrics, the percentage of classes ranked at the top reaches 24%. This percentage is much higher than the maximum percentage achieved with SBFL alone, which is 16% with Ochiai. On the contrary, we do not observe any significant improvements in the number of flaky tests ranked in the top 10 or top 5. Combined, these results imply that these change and size metrics can give additional signals that break ties between the classes located near the top, allowing developers to identify the exact cause of flakiness more precisely. The comparison with the results of GP with only SBFL formulæ in Table V further supports the usefulness of change and size metrics. Specifically, by adding change and size metrics, the percentage of flaky classes ranked at the top ($acc@1$) goes from 13% to 24% and 18%, respectively. In addition, average R_{wef} improves 5% with change metrics and 4% with size metrics.

With regard to flakiness metrics, their combination with SBFL scores does not lead to any notable improvements in the ranking of classes at the top. The percentage of classes at the top is 11% and the percentage of classes in the top 10 is 53%. One possible explanation for this is that our flakiness metrics are derived from a flakiness taxonomy that focuses on the test instead of the CUT. Hence, using metrics derived from such categories may not be helpful in the identification of CUT components that are responsible for flakiness. To alleviate this, future studies should consider categories and metrics that are derived from the CUT, and existing flakiness taxonomies should be updated accordingly.

To further investigate the impact of change and size metrics on the identification performance, we analyse the involvement of each metric in our GP-evolved formulæ. Table VII shows the frequency of change and size metrics in the GP evolved formulæ generated under the configuration of using SBFL and change metrics (*i.e.,* SBFL & Change) and the configuration of using SBFL and size metrics (*i.e.,* SBFL & Size). As shown in this table, both change and size metrics are frequently involved in the final formulæ, confirming that our observed improvement did not come only from using GP. Based on these results, we posit that change and size metrics can contribute positively to the identification of flaky classes.

TABLE VI: RQ2: The contribution of flakiness, change, and size metrics to the identification of flaky classes.

Proj. (#)	SBFL & flakiness						SBFL & change						SBFL & size					
	acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})	
	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med
Hbase (8)	1	4	5	5	11.9 (12)	3 (4)	2	4	4	5	16.9 (13)	4 (4)	2	4	5	5	11.4 (12)	3 (3)
Ignite (14)	0	2	2	4	230.9 (26)	63 (4)	2	4	4	4	222.3 (24)	18 (4)	1	3	3	5	220.1 (24)	43 (4)
Pulsar (10)	2	5	6	8	10.2 (15)	3 (8)	3	5	7	9	8.0 (12)	2 (5)	2	5	7	9	6.9 (13)	2 (6)
Alluxio (3)	0	0	1	1	97.7 (51)	73 (65)	0	0	1	1	75.7 (49)	94 (39)	0	0	1	1	90.7 (49)	77 (58)
Neo4j (3)	1	2	2	2	19.3 (42)	1 (18)	2	2	2	2	6.7 (37)	0 (9)	2	2	2	2	23.0 (40)	0 (10)
Total (38)	4	13	16	20	99.5 (24)	8 (8)	9	15	18	21	94.1 (21)	5 (6)	7	14	18	22	94.3 (22)	5 (7)
Percentage (%)	**11**	34	42	**53**	-	-	**24**	39	47	55	-	-	**18**	37	47	58	-	-

TABLE VII: Frequency of metrics in GP-evolved formulæ (from 0 to 1). 'Changes' and 'Dev' denote *'Unique Changes'* and *'Developers'*, respectively. The column 'SBFL' contains the average frequency of the four SBFL metrics.

	SBFL	Changes	Dev	Age	LOC	DOI	CC
SBFL & Change	0.45	0.50	0.37	0.53	-	-	-
SBFL & Size	0.50	-	-	-	0.71	0.37	0.73

The augmentation of Spectrum-Based Fault Localisation with change or size metrics lets more flaky classes be ranked near the top; by adding change metrics, we can rank 24% flaky classes at the top. In contrast, metrics specific to flakiness categories do not provide any beneficial signals to the identification approach.

C. RQ3 - Ensemble method

Table VIII presents the evaluation results for the voting method with 60 GP-evolved models, half from using SBFL and change metrics and the other half from using SBFL and size metrics. We decided to exclude the models that build on flakiness metrics since their usage did not improve the performance any further. As explained in Section III-C, there can be a case where none of the participating models succeeds to vote for the true candidate since individual models vote only for those ranked within the top n. For this case, we report the median of all rankings of the models as an alternative.

The results show that the voting step further improves the ranking results. The most notable improvement is the accuracy at the top 3, which reaches 47%. Although the improvements in the other accuracy metrics are not as noticeable as what we have seen in the accuracy at the top 3, there are constant improvements over the results without voting. The average of wasted effort remains almost the same while the median improves from the voting, dropping to 3.5. These results imply that the voting allows those near the top to shift further to higher ranks based on the agreement among the models that exploit and capture different features of flaky classes. Nonetheless, the constant improvements in R_{we}, both per project and in total, suggest that through the voting, we can rank flaky classes further near to the top; for example, in Alluxio, where R_{wef} is always near 50, average R_{wef} reduces to 22 and its median to 10. These results imply that voting can leverage the complementarity between different models, further improving the localisation of flakiness.

TABLE VIII: RQ3: The effectiveness of the voting between 60 different GP-evolved models, 30 from SBFL with change metrics, and 30 from using SBFL with size metrics. 'Perc' denotes Percentage

Project	Total	acc				wef (R_{wef})	
		@1	@3	@5	@10	mean	med
Hbase	8	3	5	6	6	9.62 (12)	1.5 (2)
Ignite	14	2	4	4	4	228.61 (24)	17.5 (4)
Pulsar	10	3	6	7	9	7.30 (12)	2.0 (5)
Alluxio	3	1	1	1	2	61.83 (22)	9.0 (10)
Neo4j	3	1	2	2	2	19.67 (42)	1.0 (18)
Total	38	10	18	20	23	94.61 (19)	3.5 (5)
Perc (%)	100	26	**47**	53	61	-	-

A voting between models based on SBFL, change, and size metrics, provides the best ranking for flaky classes. 47% of flaky classes are ranked in the top 3 and 26% of them are ranked at the top. The average R_{wef} further reduces to 19, highlighting the practical usefulness of our approach.

D. RQ4 - Flakiness categories

Table IX presents the performances of the voting method on the different flakiness categories encountered in our dataset. The *"Ambiguous"* category represents cases where the flaky tests could not be assigned to any of the known flakiness categories. First, we observe that the most common categories are Concurrency and Asynchronous Waits. This aligns with observations from previous studies [24], [49], [9] and confirms that the taxonomy adopted for our metrics is adequate for our distribution. Furthermore, we observe a discrepancy between the performances in different categories. Classes responsible for Async Waits are well identified with 80% of the classes in the top 10, and 30% of them at the top. Classes responsible for Concurrency also show good performances with 50% of them in the top 10, and 38% of them at the top; the average R_{wef} is below ten, eight precisely, meaning we can locate flaky classes by inspecting less than 10% of the total number of the classes covered by flaky tests.

Categories such as Time and I/O show much lower performances, with 33% and 0% of flaky classes in the top 10, respectively. Nevertheless, given the low number of instances for these categories, it is hard to discuss or generalise their results. With only two instances, the category Unordered Collections shows curious results as one class is ranked second and the other one ranked 663. To understand the reasons behind the bad ranking, we manually inspected this case[3]. We found that the concerned test, `testUnstableTopology`, was executed twice due to a retry mechanism. Both executions led to failure, but interestingly, we found that the two failures have different causes. One of them is due to a lack of context initialisation and is likely to be the reason behind flakiness. As the two failure causes are different, the coverage is also different in them. Specifically, one of the failures did not cover the flaky class, and as the coverage of this failure was leveraged in the SBFL, the flaky class was not considered suspicious. We discuss other reasons responsible for poor ranking in Section V.

TABLE IX: RQ4: The effectiveness per flakiness category

Flakiness Category	**acc** @1	@3	@5	@10	**wef (R_{wef})** mean	med
Concurrency (16)	6 (**38**)	7 (44)	7(44)	8 (**50**)	147.53 (27)	9.5 (9)
Async wait (10)	3 (**30**)	6 (60)	8 (80)	8 (**80**)	21.05 (8)	1.5 (3)
Ambiguous (4)	1 (25)	2 (50)	2 (50)	3 (75)	18.88 (5)	3.5 (5)
Time (3)	0 (0)	0 (0)	0 (0)	1 (**33**)	88.33 (16)	14.0 (10)
Network (2)	0 (0)	2 (100)	2 (100)	2 (100)	1.00 (10)	1.0 (10)
Unordered collections (2)	0 (0)	1 (50)	1(50)	1 (50)	331.5 (33)	331.5 (33)
I/O (1)	0 (0)	0 (0)	0(0)	0 (**0**)	12.50 (3)	12.5 (3)
Random (1)	0 (0)	1 (100)	1 (100)	1 (100)	2.00 (75)	2.0 (75)
Total (39[4])	10	18	20	23	94.47 (19)	3.5 (5)
Perc (%)	26	47	53	61	-	-

The most prominent flakiness categories, Concurrency and Asynchronous Waits, are identified effectively, with 38% and 30% of their flaky classes ranked at the top, respectively. In the Concurrency category, flaky classes are identified by examining 8% of classes covered by flaky tests on average.

V. Discussion

In this section, we discuss our results in light of the existing literature on test flakiness and fault localisation. Our approach uses existing fault localisation techniques to identify flaky classes in the CUT. While we leverage various data sources, the main strength of our approach comes from adopting existing SBFL techniques, as explained in RQ2. The effectiveness of other data, such as change metrics, is limited in providing additional signals that break ties between the classes already ranked near the top. Hence, the performance of our approach largely depends on the applicability of SBFL to our flaky class identification problem. The flaky class identification problem and traditional fault localisation problems are similar in the way they are debugged (*i.e.,* from the reproduction and cause identification to the fix). As described in III-A, this resemblance allows us to redefine SBFL techniques to identify flaky classes instead of faulty ones. Nevertheless, there is one significant difference between them: the characteristics of a test suite. Many fault localisation studies assume a test to cover a single functionality, and the subjects they studied often follow this assumption [26], [27], [56]. In contrast, we did not consider such an assumption for test subject selection to reflect a realistic scenario of flaky test failure. This difference may restrict the applicability of existing fault localisation techniques to the flaky class identification problem, especially test coverage-based techniques, such as SBFL. Indeed, although we identified 26% and 61% of flaky classes at the top and within the top 10, we failed to reach the performance reported in prior work on fault localisation [57]. Hence, we investigate the diagnosability of the test suite of our subjects using the Density, Diversity, and Uniqueness (DDU) metric [58].

DDU diagnoses the adequacy of SBFL for a software system by considering three properties of its test suite: Density, Diversity, and Uniqueness. Each property covers a distinct feature of a test suite, and DDU is computed as the multiplication of these three properties. Density evaluates how frequently a code entity, in our case a class, is covered by tests. Diversity is about whether tests cover code entities in a diverse fashion. Lastly, uniqueness guarantees that different code entities are covered by different sets of tests. All these three components of DDU have values between 0 and 1. The higher the DDU is, the more adequate the test suite is for SBFL.

Table X presents DDU values for the five projects analysed in this study. While all five projects generally have high diversity values (*i.e.,* all above 0.9), they have relatively low uniqueness and density values, which results in low DDU scores. Among the five projects, Pulsar has the highest DDU score of 0.289, followed by Neo4j, Alluxio, Ignite, and Hbase. Since both Neo4j and Alluxio have only three flaky classes, which might be too small to discuss the identification results, we will skip these two for the following discussion. Among the remaining three projects, all our flaky class identification methods, ranging from pure SBFL to voting, perform the best on Pulsar, the one with the highest DDU score, in *acc@n* and *wef*. For instance, even the pure SBFL approach that often performs the worst successfully localised nine out of ten flaky classes of Pulsar within the top 10 and more than half within the top five. The same trend was observed in both GP and voting-based methods. Compared to HBase, while Ignite has a slightly higher average for the DDU score, it has a far lower Uniqueness score (*i.e.,* 0.188 for Ignite and 0.413 for HBase). Uniqueness evaluates whether a code entity is distinguishable; we assume that the flaky classes have different coverage than non-flaky classes. Thus, we suspect that Ignite having a lower Uniqueness is why our methods were not as effective on Ignite as on HBase: we have the worst results on Ignite in both absolute (*i.e., acc@n* and *wef*) and relative effort (*i.e.,* R_{wef}).

[3]https://github.com/apache/ignite/commit/188e4d52c2

[4]One flaky class belongs to two categories: Network and Unordered Collections.

Based on these results, we argue that while our outcome may not be as good as those reported by prior fault localisation studies[39], [40], that is mainly due to the inherently low diagnosability of a test suite (*e.g.,* covering too many classes in the same fashion). This test-suite adequacy issue commonly exists in the fault localisation field[51] and is not limited to flaky class identification. Hence, we posit that the performance of our approach can improve along with the advances in fault localisation techniques.

In an attempt to shed light on the 15 cases where the class was ranked outside the top 10 by our voting approach, we extended our inspection to reason about such performances. We observed that flaky tests covering a high number of classes are more likely to result in low performances. For example, the flaky test `shutdownDatabaseDuringIndexPopulations` in Neo4j covers 480 classes and its flaky class was ranked 59 by our voting approach whereas the other flaky tests in Neo4j (having their corresponding flaky classes ranked 1 and 2) cover fewer than 10 classes. When we inspect the DDU score of the specific commit that contains this test, it has a relatively low DDU score compared to the other two commits. Additionally, most of the mis-ranked classes are found in the Ignite project (10/15), whose DDU score is the second-lowest, and its tests cover on average 492 classes. Due to this consequent number of covered classes, we suspect these tests to be of a higher level, *i.e.,* integration or end-to-end tests. This aligns with studies highlighting the prevalence of flakiness in integration and system tests [59], [60]. Still, our approach does not systematically fail to identify flaky classes covered by higher-level tests as nine of them (flaky test covering more than 100 classes) are listed in the top 10.

VI. Threats to Validity

a) External validity: The main threat to the external validity of this study is the dataset size. To ensure the generalisability of our results, it would have been preferable to include more flaky tests in our experiments. Nonetheless, the datasets of flaky tests are generally limited in size due to the elusiveness of flakiness [61], [14], [13]. Moreover, as explained in Section II, the requirements of this study limited the set of candidates considerably. For a commit to be eligible in our study, it needs to have atomic changes fixing flakiness in the CUT. However, only 24% of flaky tests actually stem from the CUT, which limits the size of potential subjects [24]. Besides, the creation of our dataset required a substantial amount of manual work to identify suitable commits and perform necessary changes to retrieve coverage matrices. For instance, for each commit, we had to modify the build script to match GZOLTAR requirements, *i.e.,* find the test executor version that matches both the program under test and the plugin. We iteratively removed non-essential listeners and other plugins that could interfere with the instrumentation. Moreover, we had to find and adapt the execution environment to match the program under test and the testing environment. Finally, compared to the works of Lam *et al*. [62] and Zitfci and Cavalcanti [63], which were conducted on proprietary software, this study is the first to leverage open-source software to localise flakiness root causes. Thus, our dataset and ground truth can be valuable for future studies on flakiness debugging.

b) Internal validity: One potential threat to our internal validity is our definition of flakiness root causes within the CUT, *i.e.,* flaky classes. We rely on flakiness-fixing commits to identify classes that are responsible for flakiness. However, we cannot be certain that (i) the flakiness fix is effective, and (ii) the modified class is the one responsible for flakiness. Indeed, a study by Lam *et al*. [49] showed that developers may wrongly claim that their commits fix flaky tests before realising that the fix is ineffective. Additionally, there are no guarantees that the classes included in the fix are the ones responsible for flakiness. Nonetheless, if the class was part of the proclaimed fix, this means that the developers found it, at least, relevant. Hence, its identification by our approach is still helpful for developers willing to understand, debug, and fix flaky tests.

c) Construct validity: One potential threat to our construct validity is our measurement of the coverage for flaky tests. A flaky test can pass and fail for the same version of the program, but in practice, it may be extremely difficult to reproduce both the pass and failure [13], [64]. Hence, a test can be observed as flaky by the project developers and therefore fixed, yet we are unable to reproduce the pass and failure in our experiments even with a large number of reruns [13]. For this reason, we focused on the available status, *i.e.,* pass or failure, and retrieve its coverage. It is possible that including the coverage of both the pass and failure from the flaky tests might lead to different results with spectrum-based fault localisation. Thus, we encourage future studies to investigate this direction. Another possible threat is whether the evaluation results of our approach truly support what we claim. We use two absolute metrics, $acc@n$ and wef, that can reflect the realistic debugging effort of developers, following the suggestion from Parnin and Orso [54], and one relative metric, R_{wef}, to compare with the baseline of inspecting classes covered by flaky tests.

VII. Related Work

a) Flakiness root causes: Several empirical studies highlighted the diversity of flakiness root causes. Luo *et al*. [24] were the first to characterise the root causes of flaky tests. They analysed 201 flakiness-related commits from 51 open-source projects and showed that the mismanagement of asynchronous calls and concurrency are the most common causes of flaky tests. Later studies replicated the work of Luo *et al*., showing that other flakiness root causes can be more relevant in different application domains. Thorve *et al*. [53] analysed 77 flakiness-related commits in 29 open-source Android applications and found that 22% of these commits have flakiness caused by external factors like hardware, operating system version, and third-party libraries. Eck *et al*. [9] surveyed 21

TABLE X: DDU metrics for the analysed test suites.

Project	Density			Diversity			Uniqueness			DDU		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Hbase	0.049	0.477	0.248	0.995	0.999	0.997	0.188	0.553	0.413	0.021	0.116	0.091
Ignite	0.368	0.993	0.736	0.918	1.000	0.979	0.045	0.486	0.188	0.034	0.466	0.132
Pulsar	0.029	0.998	0.491	0.984	0.998	0.994	0.520	0.786	0.609	0.019	0.518	0.289
Alluxio	0.414	0.833	0.591	0.958	0.996	0.982	0.226	0.615	0.362	0.101	0.322	0.201
Neo4j	0.127	0.739	0.515	0.894	0.993	0.931	0.268	0.791	0.585	0.088	0.522	0.258

Mozilla developers, asking them to classify 200 flaky tests in terms of root causes and fixing efforts. The survey results highlighted four new categories of flakiness: restrictive ranges, test case timeout, test suite timeout, and platform dependency.

b) Flakiness root cause analysis: The main contribution to flakiness root cause localisation was proposed by Lam *et al*. [62]. They introduced a framework that helps developers to localise the root causes of their flaky tests. This framework uses an instrumentation tool to log the runtime properties of the test execution. Then it reruns the tests 100 times to produce logs for a passing and a failing execution. To analyse these logs and localise the root cause, they propose RootFinder, a tool that compares the logs of passing and failing executions to identify methods that can be responsible for flakiness. RootFinder relies on a predefined set of non-deterministic method calls and does not explore calls of unknown methods. Hence, it can only detect flaky tests that arise from method calls that the developer is already suspecting. Zitfci and Cavalcanti [63] presented Flakiness Debugger, a tool that compares the code coverage of passing and failing executions to localise the flakiness root cause. They ran their tool on 83 flaky tests and presented the localised root cause to two developers asking them for their evaluation. On average the developers found that in 48% of the cases, flakiness was due to the exact statements spotted by Flakiness Debugger. Moreover, only 18% of the outputs were considered inconclusive, hard to understand, or not useful. Both RootFinder and Flakiness Debugger relied on differences between passing and failing executions of flaky tests to localise flakiness in the CUT. In this study, we explore a new direction by analysing the differences between flaky and stable tests.

Morán [65] presented FlakyLoc, a tool for localising the root causes of flakiness in web applications. The tool reruns web tests while varying environmental factors (network, memory, CPU, browser type, operating system, and screen resolution) and records test results. Then, it uses ranking metrics (Ochiai and Tarantula [66], [67]) to identify the environmental factor and value that are responsible for the flaky failure. The tool was only evaluated on one test case and it detected that the failure was caused by low screen resolution. In this paper, we do not focus on any specific flakiness category and our analysis is based on the test coverage instead of environmental factors.

c) Fault localisation: Fault localisation was introduced to ease the developers' burden of debugging by automatically identifying the root cause of a program failure [68], [57]. Since its appearance, various fault localisation techniques that utilise various data, from dynamic to static ones, have been actively proposed [69], [29], [27], [32]. Spectrum Based Fault Localisation (SBFL), a lightweight coverage-based technique, was under the spotlight for many years. SBFL takes a test coverage matrix as input and computes the likelihood of containing a fault for individual code entities using a risk evaluation formula. The simplicity and effectiveness of SBFL attract many researchers into this field [70], [71], [72], [39]. Papadakis and Traon proposed Mutation Based Fault Localisation (MBFL) techniques that leverage the coupling between real faults (*i.e.,* complex faults) and the mutants (*i.e.,* simple faults) to localise faults in code [29]. Information Retrieval-based Fault Localisation (IRFL) approaches the problem differently, utilising static data sources, such as bug reports, instead of dynamic ones, such as test coverage [32]. Recently, Li *et al*. proposed to combine various fault localisation techniques using a deep learning model [27]. Their approach called DeepFL successfully outperforms all the other FL techniques that it considers. The current trend of fault localisation is moving to either use a deep neural network to train a model [26], [73] or include humans to bring additional signals [74]. In which direction it heads, the main framework of fault localisation remains the same and test coverage remains to be an effective source of information.

VIII. Conclusion

We presented the first empirical evaluation of SBFL as a potential approach for identifying flaky classes. We investigated three approaches: pure SBFL, SBFL augmented with change and code metrics, and an ensemble of them. We evaluated these approaches on five open-source Java projects. Our results show that SBFL-based approaches can identify flaky classes relatively well, especially with code and change metrics, suggesting that code components responsible for flakiness exhibit similar properties with faults. This finding highlights the potential of existing fault localisation techniques for flakiness identification. At the same time, the results show that flaky tests can have unique failure causes that may mislead any coverage-based root cause analysis, stressing the need to consider these flakiness-specific causes in future studies.

Our study forms the first step towards flakiness localisation. We believe that there is a lot of room for improvement and encourage future studies to explore additional techniques, fault prediction metrics, and devise techniques that can further improve and support flakiness localisation.

REFERENCES

[1] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, H. Sharp and M. Whalen, Eds. IEEE / ACM, 2019, pp. 101–110. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00019

[2] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. L. Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," pp. 244–255, 2022.

[3] G. F. Martin Gruber, "A survey on how test flakiness affects developers and what support they need to address it," *International Conference on Software Testing (ICST)*, 2022.

[4] J. Micco, "The State of Continuous Integration Testing Google," 2017.

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

[6] J. Palmer, "Test flakiness – methods for identifying and dealing with flaky tests : Spotify engineering," https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/, November 2019, (Accessed on 01/12/2021).

[7] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 857–862, 2018.

[8] J. Micco and A. Memon, "Gtac 2016: How flaky tests in continuous integration - youtube," https://www.youtube.com/watch?v=CrzpkF1-VsA, December 2016, (Accessed on 01/12/2021).

[9] M. Eck, M. Castelluccio, F. Palomba, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," *arXiv*, pp. 830–840, 2019.

[10] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pp. 312–322, 2019.

[11] D. Silva, L. Teixeira, and M. D'Amorim, "Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker," *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 301–311, 2020.

[12] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. New York, New York, USA: ACM Press, 2018, pp. 433–444. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3180155.3180164

[13] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1572–1584.

[14] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests," *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021.

[15] G. Pinto, B. Miranda, S. Dissanayake, M. D'Amorim, C. Treude, and A. Bertolino, "What is the Vocabulary of Flaky Tests?" *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pp. 492–502, 2020.

[16] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in Android via event order exploration," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23â•fi28, 2021, Athens, Greece*, vol. 1, no. 1. Association for Computing Machinery, 2021, pp. 367–378.

[17] B. Camara, M. Silva, A. T. Endo, and S. Vergilio, "What is the vocabulary of flaky tests? an extended replication," in *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 444–454. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICPC52881.2021.00052

[18] B. Camara, M. Silva, A. Endo, and S. Vergilio, "On the use of test smells for prediction of flaky tests," in *Brazilian Symposium on Systematic and Automated Software Testing*, 2021, pp. 46–54.

[19] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A Black-Box, Language Model-based Predictor for Flaky Tests," *arXiv preprint arXiv:2112.12331*, pp. 1–12, 2021. [Online]. Available: http://arxiv.org/abs/2112.12331

[20] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies : A Framework for Automatically Fixing Order-Dependent Flaky Tests," in *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations ofSoftware Engineering (ESEC/FSE '19)*, 2019.

[21] S. Dutta, A. Shi, and S. Misailovic, *FLEX: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds*. New York, NY, USA: Association for Computing Machinery, 2021, p. 603–614. [Online]. Available: https://doi.org/10.1145/3468264.3468615

[22] M. Gruber, S. Lukasczyk, F. Krois, and G. Fraser, "An Empirical Study of Flaky Tests in Python," *Proceedings - 2021 IEEE 14th International Conference on Software Testing, Verification and Validation, ICST 2021*, pp. 148–158, 2021.

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

[24] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, nov 2014, pp. 643–653.

[25] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[26] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 664–676. [Online]. Available: https://doi.org/10.1145/3468264.3468580

[27] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," 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, 2019, p. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[28] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 2007, pp. 137–146.

[29] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Journal of Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[30] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 464–475.

[31] A. Perez, R. Abreu, and I. HASLab, "Leveraging qualitative reasoning to improve sfl." in *IJCAI*, 2018, pp. 1935–1941.

[32] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[33] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[34] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[35] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," University College London, Tech. Rep. RN/14/14, 2014.

[36] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 191–200.

[37] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 177–188.

[38] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[39] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 4:1–4:30, July 2017.

[40] J. Sohn and S. Yoo, "Empirical evaluation of fault localisation using code and change metrics," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1605–1625, 2021.

[41] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, July 2012.

[42] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *The proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC 2006. IEEE, 2006, pp. 39–46.

[43] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 88–99.

[44] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *Proceedings of ICSE Workshop on Software Visualization*, 2001, pp. 71–75.

[45] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 467–477.

[46] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018.

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

[48] O. Parry, "A Survey of Flaky Tests," *ACM transactions on software engineering and methodology*, vol. 31, no. 1, 2021.

[49] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," *Proceedings - International Conference on Software Engineering*, pp. 1471–1482, 2020.

[50] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document

[51] J. Sohn, G. An, J. Hong, D. Hwang, and S. Yoo, "Assisting bug report assignment using automated fault localisation: An industrial case study," in *Proceedings of the 14th IEEE International Conference on Software Testing, Verification and Validation*, 2021.

[52] J. Sohn and S. Yoo, "Why train-and-select when you can use them all? Ensemble model for fault localisation," in *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO 2019, 2019, pp. 1408–1416.

[53] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pp. 534–538, 2018.

[54] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 199–209. [Online]. Available: https://doi.org/10.1145/2001420.2001445

[55] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 06, pp. 803–827, 2011.

[56] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S. C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[57] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[58] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 654–664.

[59] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at apple," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 110–119. [Online]. Available: https://doi.org/10.1145/3377813.3381370

[60] K. Herzig and N. Nagappan, "Empirically Detecting False Test Alarms Using Association Rules," *Proceedings - International Conference on Software Engineering*, vol. 2, pp. 39–48, 2015.

[61] S. Habchi, M. Cordy, M. Papadakis, and Y. L. Traon, "On the use of mutation in injecting test order-dependency," *CoRR*, vol. abs/2104.07441, 2021. [Online]. Available: https://arxiv.org/abs/2104.07441

[62] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root Causing Flaky Tests in a Large-Scale Industrial Setting," in *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. Beijing, China: ACM Press, 2019, pp. 101–111.

[63] 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)*, 2020, pp. 736–745.

[64] 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)*. IEEE, 2020, pp. 403–413.

[65] J. Morán, C. Augusto, A. Bertolino, C. de la Riva, and J. Tuya, "Flakyloc: Flakiness localization for reliable test suites in web applications," *J. Web Eng.*, vol. 19, no. 2, pp. 267–296, 2020. [Online]. Available: https://doi.org/10.13052/jwe1540-9589.1927

[66] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, p. 1780–1792, nov 2009. [Online]. Available: https://doi.org/10.1016/j.jss.2009.06.035

[67] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 273–282. [Online]. Available: https://doi.org/10.1145/1101908.1101949

[68] C. Catal, "Software fault prediction: A literature review and current trends," *Expert Systems with Applications*, vol. 38, no. 4, pp. 4626 – 4636, 2011.

[69] W. Wen, "Software fault localization based on program slicing spectrum," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1511–1514.

[70] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, ser. COMPSAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 449–456.

[71] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering Methodology*, vol. 22, no. 4, pp. 31:1–31:40, October 2013.

[72] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, G. Ruhe and Y. Zhang, Eds. Springer Berlin Heidelberg, 2013, vol. 8084, pp. 224–238.

[73] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 661–673. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00067

[74] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 82–92. [Online]. Available: https://doi.org/10.1145/3180155.3180242