Quantifying, Characterizing, and Mitigating Flakily Covered Program Elements

Shivashree Vysali, Shane McIntosh, *Member, IEEE,* and Bram Adams, *Member, IEEE*

Abstract—Code coverage measures the degree to which source code elements (e.g., statements, branches) are invoked during testing. Despite growing evidence that coverage is a problematic measurement, it is often used to make decisions about where testing effort should be invested. For example, using coverage as a guide, tests should be written to invoke the non-covered program elements. At their core, coverage measurements assume that invocation of a program element during any test is equally valuable. Yet in reality, some tests are more robust than others. As a concrete instance of this, we posit in this paper that program elements that are only covered by flaky tests, i.e., tests with non-deterministic behaviour, are also worthy of investment of additional testing effort. In this paper, we set out to quantify, characterize, and mitigate "flakily covered" program elements (i.e., those elements that are only covered by flaky tests). To that end, we perform an empirical study of three large software systems from the OpenStack community. In terms of quantification, we find that systems are disproportionately impacted by flakily covered statements with 5% and 10% of the covered statements in Nova and Neutron being flakily covered, respectively, while $< 1\%$ of Cinder statements are flakily covered. In terms of characterization, we find that incidences of flakily covered statements could not be well explained by solely using code characteristics, such as dispersion, ownership, and development activity. In terms of mitigation, we propose GreedyFlake – a test effort prioritization algorithm to maximize return on investment when tackling the problem of flakily covered program elements. We find that GreedyFlake outperforms baseline approaches by at least eight percentage points of Area Under the Cost Effectiveness Curve.

Index Terms—Code coverage, Software testing, Flaky tests

1 Introduction

Code coverage tools measure how thoroughly tests exercise programs [1]. By instrumenting a program during test suite execution, code coverage tools determine which program elements have been invoked and which ones have not. Coverage reports provide an overview of the proportion of all program elements that have been invoked during testing [1]. Although they may target program elements at varying granularities (e.g., statements, branches), their essential mode of operation remains the same.

Since low code coverage indicates that plenty of program elements have not been tested, it is common practice for software organizations to use coverage measurements as a quality gate in their integration pipelines. For example, the Apache Software Foundation has a quality gate that enforces a minimum code coverage of 80% by default.[1] Changes that do not meet this quality criterion are blocked from integration into the product.

Conversely, it is assumed that high coverage indicates adequate testing. Goodhart's law (a popular adage) states that "When a measure becomes a target, it ceases to be a good measure" [2] – this is indeed true of coverage measurements. Fowler has argued that when coverage improvements are targeted, developers tend to focus on writing tests that improve coverage, rather than writing tests that can catch defects.[2] This increases the cost of test execution and maintenance by adding additional tests; however, the benefits in terms of test suite effectiveness are unclear. Indeed, studies of the relationship between coverage and test suite effectiveness have produced mixed results [3], [4].

At their core, coverage measurements are based on a coarse-grained binary classification of program elements. Elements are either labelled as invoked during testing or not. It is our position that this classification is limiting the value of coverage measurements. Expanding the classification to a broader set of categories may yield more actionable insights. For example, coverage can be classified based on the scope of the covering test (e.g., unit, integration). Coverage by one scope may not imply coverage by another. Program elements can also be covered by *flaky tests*, i.e., tests that exhibit non-deterministic behaviour. Program elements that are covered by flaky tests are unlikely to be as well-tested as program elements that are covered by tests with deterministic behaviour.

In this paper, we set out to study these flakily covered elements as well as their impact "in the wild." To do so, we perform an empirical study of the Nova, Neutron, and Cinder projects – the three largest and most active projects in the OpenStack community. We structure our empirical study along three dimensions:

1) **Quantification**: To what degree are program elements flakily covered?
Motivation: While a fine grained coverage analysis has benefits, the additional costs involved in

- *Shivashree Vysali is with the Department of Electrical and Computer Engineering, McGill University, Canada.
E-mail: shivashree.vaidhyamsubramanian@mail.mcgill.ca*
- *Shane McIntosh is with the David R. Cheriton School of Computer Science, University of Waterloo, Canada.
E-mail: shane.mcintosh@uwaterloo.ca*
- *Bram Adams is with the School of Computing, Queen's University, Canada.
E-mail: bram.adams@queensu.ca*

1. https://sonarcloud.io/organizations/apache/quality_gates
2. https://martinfowler.com/bliki/TestCoverage.html

© 2020 IEEE. Author pre-print copy. The final publication is available online at: https://dx.doi.org/10.1109/TSE.2020.3010045

determining flakily covered program elements needs to be justified. If flakily covered program elements are rare, their impact on code coverage measurements could be dismissed as negligible and the additional cost involved is not justified. Therefore, it is necessary to quantify flakily covered program elements to determine if the magnitude of flakily covered program elements is high enough to justify further analyses.

Results: We find that systems are disproportionately impacted by flakily covered statements with 5% and 10% of the covered statements in Nova and Neutron being flakily covered, respectively, while <1% of Cinder statements are flakily covered.

2) **Characterization**: Which kinds of program elements are flakily covered?
Motivation: If the occurrence of flakily covered statements can be explained by basic code characteristics, such as the location within the system, ownership, age and churn, then there is no need to distinguish between flakily covered and covered program elements.

Results: We take the position of a devil's advocate and analyze flakily covered statements along three dimensions of basic code characteristics, namely (A) Dispersion; (B) Ownership; and (C) Development activity. From our analyses, we conclude that the occurrence of flakily covered statements cannot be well explained solely by using basic code characteristics.

3) **Mitigation**: How should repair effort be prioritized?
Motivation: Software teams often operate under tight time and budget constraints. Thus, it would be useful to prioritize the mitigation of flakily covered program elements such that teams receive the largest return on investment as quickly as possible.

Results: We propose GreedyFlake, a greedy approach to prioritize the repair of flaky tests such that the ones that are associated with the largest number of flakily covered statements are fixed first. To evaluate GreedyFlake, we plot Alberg diagrams (a.k.a., lift charts) of the cost (in terms of flaky tests to be repaired) against the effectiveness (flakily covered program elements that have been repaired). We observe that GreedyFlake outperforms random and traditional test case prioritization approaches by at least eight percentage points. However, we find that there is only marginal benefit to the greedy re-ranking step (1-3 percentage points), so "greediness" is not necessary to achieve most of the benefit.

2 Related Work

In this section, we discuss the related work with respect to code coverage, test reliability and test prioritization.

2.1 Code Coverage

Code coverage is a well established concept in software engineering research and practice. Piwowarski et al. [5] explained that IBM used code coverage in the late 1960s. Marick [6] warns that "requiring" very high coverage might lead to tests being written only to satisfy coverage conditions and not to reveal bugs. Elbaum et al. [7] studied the impact of software evolution on code coverage and determined that even small changes during the evolution of a program can have a profound impact on coverage information.

As code coverage criteria are often used to evaluate test suites, many studies focus on the relationship between code coverage and test suite effectiveness. Some studies have shown that generating test suites to satisfy code coverage criteria has a positive effect on finding faults [4], [8], [9] while other studies do not [3], [10], [11], [12]. Schwartz et al. [13] investigated the faults that are missed by test suites with high coverage scores and found that they often miss faults that corrupt internal state.

Broadly speaking, most prior work has focused on understanding code coverage with respect to different granularities of program elements and exploring the risks associated with using high coverage as a quality gate. We instead propose to explore code coverage with an awareness of test characteristics and to categorize covered program elements based on test characteristics, to obtain more actionable insights from code coverage. Wong et al. [14] proposed an approach to calculate the risk of a statement based on the number of successful and failed tests that cover it. Their approach was successful in a fault localization scenario. Our approach aims to categorize covered program elements based on test reliability (flakiness).

2.2 Flaky Tests

Previous studies on flaky tests have focused on understanding the root causes of flaky tests. Luo et al [15] analyzed 201 commits in the Apache ecosystem that fixed flaky tests and reported that the three main causes of non-determinism in tests are asynchronous waits, concurrency and test order dependencies. Thorve et al. [16] performed a similar analysis for Android applications and reported three other root causes, namely, Dependency, Program Logic, and UI.

The common practice to determine if a test is flaky is to repeat the test a number of times and mark the test as flaky if the result changes. Since repeating tests is expensive, many studies have focused on automatically detecting flaky tests. Bell et al. [17] proposed an automated approach called DeFlaker, which monitors the coverage of code changes and marks as flaky any newly failing test that did not execute any of the changed lines of code. Their approach was able to detect 87 unknown flaky tests in ten active projects. Lam et al. [18] proposed an automated approach to detect order-dependent flaky tests. King et al. [19] proposed an approach that leverages Bayesian networks for classifying flaky tests.

In this paper, we rely on test execution history to build a corpus of flaky tests, then use this data to identify flakily covered program elements.

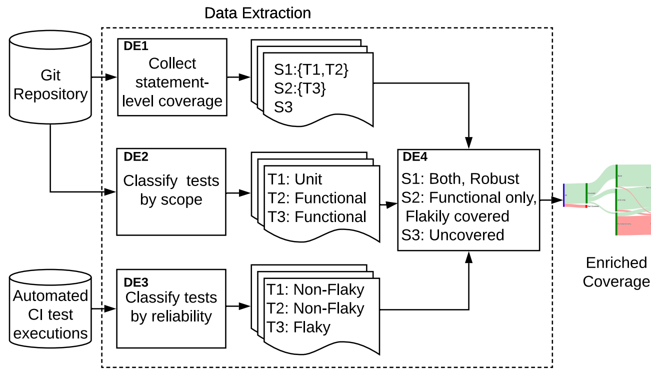


Fig. 1: An overview of data extraction

2.3 Test Case Prioritization

Test case prioritization is a means to achieve target objectives in software testing by reordering the execution sequences of test suites. Rothermel et al. [20] formally defined the test case prioritization (TCP) problem, presented several techniques for prioritizing test cases, and presented the results of empirical studies in which those techniques were applied to various programs. In particular, four coverage-based greedy test prioritization approaches were proposed. Elbaum et al. [21] extended the empirical study of Rothermel et al. by including more programs and prioritization techniques. Do and Rothermel [22] applied coverage-based prioritization techniques to the JUnit testing environment and showed that prioritized execution of JUnit test cases improved the fault-detection rate.

Greedy algorithms have also been explored. For example, Jones and Harrold [23] proposed a greedy variant to the Modified Condition/Decision Coverage (MC/DC) criterion for prioritization. Moreover, Li et al. [24] compared random prioritization and a genetic test case prioritization algorithm with several greedy algorithms. They observed that greedy algorithms are often outperformed by optimal algorithms, but the simplicity and cost effectiveness of greedy algorithms still merits their usage. Given their promising results, we propose a greedy approach to tackle flakily covered program elements.

Prioritization approaches may also focus on which areas of the codebase should be improved first. For example, Shihab et al. [25] leveraged the development history of a project to generate a prioritized list of functions to focus unit test writing resources on. In our work, we obtain a prioritized list of flaky tests to minimize flakily covered program elements.

3 Study Design

In this study, we set out to analyze code coverage with an awareness of characteristics of the test(s) that cover(s) each statement in the source code. Specifically, we study test scope and reliability characteristics. In this section, we outline our approach for collecting the data required to analyze coverage from different perspectives.

3.1 Studied Systems

In order to analyze code coverage with an awareness of test characteristics, we need projects with a clearly defined testing process. Therefore, we focus on projects from the OpenStack community for analysis. The OpenStack community has (a) clear testing guidelines for its projects and (b) a robust continuous integration system with test execution results available for submitted patches.

We need large and active projects, to maximize our chances of observing flaky tests. We start by identifying projects that form the core of OpenStack,[3] namely Nova, Neutron, Cinder, Keystone, Glance, Swift and Horizon.

Next, we need to ensure that we are able to collect complete coverage information by running the test suites successfully. Most OpenStack projects use Tox to install the dependencies needed for testing.[4] Using this Tox environment, we could successfully replicate the testing environments for Nova, Neutron and Cinder.

3.2 Data Extraction

Figure 1 provides an overview of the steps involved in the coverage and test characteristics data extraction process.

DE1: Collect statement-level coverage

We first need to compute a test-to-statement mapping, i.e., a many-to-many relation where each statement may be covered by zero or more tests and each test may cover zero or more statements. The main purpose of the test-to-statement mapping is to enable fine-grained analysis. Since we set out to analyze scope- and reliability-aware coverage perspectives, this mapping is a critical data structure upon which we will build.

Since our studied projects are implemented in Python, we use Coverage.py,[5] a popular Python coverage tool, to collect coverage at the statement level. The result is a Coverage database (CovDB), which contains a list of all the statements executed during coverage collection and a test-to-statement mapping.

Recent work by Shi et al. [26] demonstrated that flaky tests can yield unreliable coverage measurements. To mitigate the risks posed by flaky tests, for each project, we repeat the collection of coverage measurements ten times. In our coverage collection scenarios, we did not observe any test failures. In fact, we found that the coverage measurements are stable and do not change across the ten runs. We do not believe this is irregular, as Shi et al. found that coverage instability was project-sensitive.

Another concern is accurate test-to-statement mapping when tests share setup/teardown code. The studied projects use the unittest framework for testing, which supports sharing setup/teardown methods both at the test case level and test class level.[6] When code is shared at the test case level (using setUp/tearDown methods), the unittest framework executes the shared statements for each test case, which allows Coverage.py to map shared statements to all the tests

3. https://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html
4. https://tox.readthedocs.io/en/latest/
5. https://coverage.readthedocs.io/en/coverage-5.1/
6. https://docs.python.org/3/library/unittest.html

that execute them. When code is shared at the test class level (using setUpClass/tearDownClass methods), Coverage.py does not map the code to individual tests in the class. However, we do not find any instances of shared code at test class level in the studied projects.

DE2: Classify tests by scope

Tests are written with different intended scopes. For example, unit tests focus on isolating the smallest modules for individual testing, while integration or functional tests target logical groups of modules or system-level functionality. Since coverage by one scope may not imply coverage by another (e.g., integration-level issues cannot be discovered by unit tests), we set out to study how coverage varies with respect to scope.

In the studied projects, tests are organized based on their scope into separate folders (unit and functional). We determine the scope of each test by analyzing the code base directory in which the test appears.

DE3: Update Test Flakiness

Flaky tests are tests that exhibit non-deterministic behaviour, i.e., the test results may change when the code under test has not. Flaky tests are an example of unreliable tests. Since the outcome of flaky tests is unreliable, the statements covered only by flaky tests should not raise the confidence of development teams as much as statements covered by robust tests.

Previous studies [15], [17] have relied on re-running tests several times to determine flaky tests. However, the re-execution of tests is computationally expensive. In order to avoid re-running tests, we rely on previous test execution history available through OpenStack's Continuous Integration (CI) system.

If an OpenStack developer suspects that a test result is flaky, they can request for tests to be re-executed against a specific patch. If tests are re-run against the same patch and the test result changes, it indicates the presence of a flaky test. We filter patches against which tests were run more than once and identify patches with inconsistent test outcomes. We then parse the test suite results to identify the actual test cases with non-deterministic behavior.

DE4: Categorize statements

To obtain an enriched coverage report, we categorize statements based on the characteristics of the test(s) that cover(s) the statements.

To do so, we first categorize statements based on the scope of the tests that cover the statement, using a combination of the test-to-statement mapping (DE1) and the detected scope of tests (DE2). Since statements may be covered by multiple tests, it is possible for a statement to be covered by:

- Unit tests only: The statement is covered by one or more unit tests, but no functional tests.
- Functional tests only: The statement is covered by one or more functional tests, but no unit tests.
- Both unit and functional tests: The statement is covered by at least one unit test and at least one functional test.

For each category, we further classify the statements based on the reliability of the tests covering the statement. If all of the tests covering a statement are flaky, the statement is considered as flakily covered. If there is at least one non-flaky test covering a statement, then the statement is considered robustly covered. If a statement is not covered by any test, it is considered not covered.

4 Enriched Coverage Observations

Following the procedure to categorize statements (DE4), we obtain an enriched coverage report. This report shows the total coverage for each project and splits the coverage numbers based on test scope and test reliability. We visualize the coverage split using a Sankey diagram [27]. Sankey diagrams are variants of flow diagrams, in which the width of arrows is proportional to flow quantity.

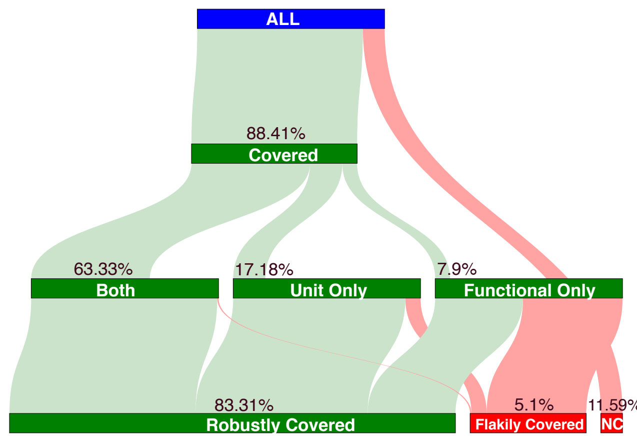
Figure 2 shows the three-tiered Sankey diagrams generated for the studied projects. At the first level, all of the statements are categorized as either covered or uncovered. At the second level, all of the covered statements are categorized based on test scope (unit only, functional only, both). At the third level, statements in each test scope category are further categorized based on test reliability (flakily-covered, non-flakily covered).

Figure 2 shows that more statements are covered only by unit tests than only by functional tests. This is not surprising because unit tests account for a larger proportion of the test suites of the subject systems (63%-95%). More interestingly, 60.94% and 63.33% of statements are covered by both unit and functional tests in Neutron and Nova, respectively, while only 30% are covered by both types of tests in Cinder. We suspect this discrepancy is caused by the lower proportion of functional tests in Cinder (5%).

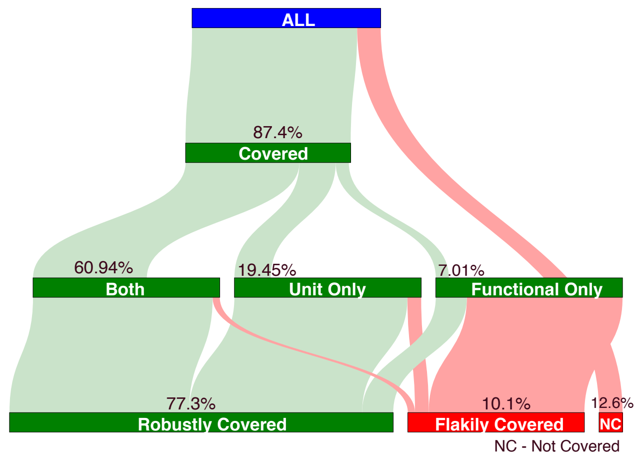
Figure 2 also shows that Cinder has the lowest coverage at 75.11%. On the surface, Nova and Neutron appear to be more thoroughly tested than Cinder. However, Cinder has the lowest percentage of flakily covered statements at 0.14%. If we were to remove the flakily covered statements from the set of covered statements, the coverage of Neutron and Cinder becomes comparable. This further supports the claim that higher coverage scores do not always indicate more thorough testing.

The Sankey diagrams help developers identify possible weakly covered statements. For example, from the Sankey diagram for Neutron, it can be seen that a large portion of statements that are covered by functional tests are flakily covered. Instead of focusing on improving code coverage numbers, developers can focus on fixing flakiness in these functional tests to improve test reliability.

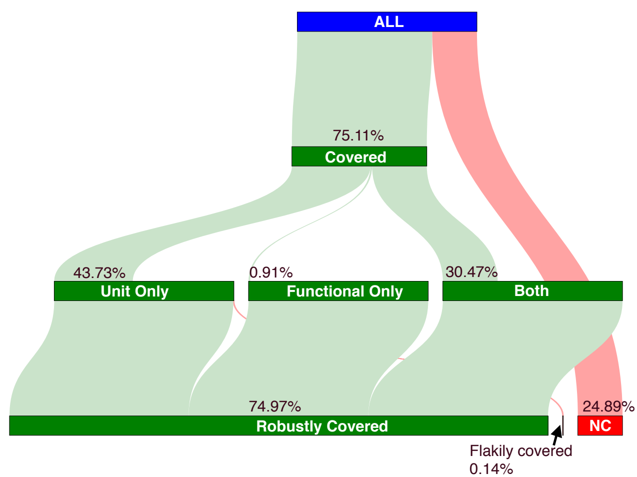
Summary of Key Findings: Our enriched coverage reports provide insights into test scope and robustness that plain coverage reports may miss. For example, Cinder, despite having lower overall coverage (75%), has the lowest proportion of flakily covered statements (0.14%). On the other hand, Neutron has higher coverage (87%) but also has the largest proportion of flakily covered statements (10%).



(a) Nova



(b) Neutron



(c) Cinder

Fig. 2: Sankey diagrams, visualization of generated enriched coverage reports. We observe that, for example, in the case of Neutron, a large (10.1%) proportion of statements that are only covered by functional tests are flakily covered.

5 Advocatus Diaboli

In this section, we explore the position of an Advocatus Diaboli (AD, i.e., a devil's advocate) to determine if flakily covered statements could be attributed to basic code, developer, or maintenance characteristics. We focus on intuitive, general code characteristics that do not involve program- or language-specific code analyses. The rationale for this choice being that if flakiness in code coverage can be tackled through general code characteristics, teams can act upon our insights without requiring expensive additional analyses. Broadly speaking, the arguments of a pragmatic AD fit into (A) Dispersion; (B) Ownership; and (C) Development Activity dimensions. For each argument, we present its rationale, our approach to evaluating it, and the results that we observed.

A. Dispersion

Dispersion properties measure the diffusion of a phenomenon across modules of the codebase. A naïve explanation of our results may be that the flakily covered statements: (A.1) are concentrated in one area of the system; (A.2) appear in poorly tested modules; or (A.3) are introduced by a small number of contributors. Below, we explore each of these AD arguments.

Argument A.1: The flakily covered statements are all part of the same module.

Rationale: The nature of some modules may increase the likelihood of tests to be flaky. For instance, a module that focuses on networking may be prone to flakily covered statements due to tests depending upon responses received from across a network. If such a naïve explanation were true, the value of our observation about the frequency of flakily covered statements may be limited.

Approach: We use treemaps [28] to investigate the concentration of flakily covered statements across modules. Treemaps allow shape nesting, size, and shade properties to be mapped on to data properties. In our treemaps, each node (box) corresponds to a source code file. Thicker lines indicate module groupings, i.e., files nested within thick lines appear within the same module. Each file in the treemap is shaded according to the number of flakily covered statements it contains (darker shaded files indicate more flakily covered statements).

Results: Figure 3 shows that flakily covered statements are often dispersed across modules. In Nova and Neutron, 70% and 79% of modules contain at least one flakily covered statement. Among those modules that contain flakily covered statements, the Nova and Neutron modules respectively have: (a) medians of 17 and 7 flakily covered statements; and (b) standard deviations of 173 and 246 flakily covered statements. Indeed, the results indicate that flakiness impacts a large proportion of modules.

On the other hand, flakily covered statements in the Cinder system are more concentrated. Figure 3(c) shows that only 17% of the modules contain at least one flakily covered statement. One plausible explanation for the higher concentration of flakily covered statements in the Cinder system may be the fact that there are only 155 identified flakily covered statements. 115 of the 155 flakily covered statements (73%) are located in the `volumes/drivers`

module – the module that contains 75% of the statements in the Cinder codebase.

While the module-level dispersion of flakily covered statements is often quite high, Figure 3 shows that some files have a larger amount of flakily covered statements than others. We observe that most of these "hotspots" are among the largest files in the module. For example, `virt/libvirt/driver.py` and `compute/manager.py` files are the largest in the `virt/libvirt` and `compute` modules in the Nova system. On further examination of `virt/libvirt/driver.py`, it appears that the file contains code to connect and configure multiple external services. Luo et al. [15] found that network dependencies were common causes of non-determinism in tests.

Closer inspection of the flakily covered statements in these hotspot files reveals that they may be especially susceptible to turbulent network conditions or incorrect platform assumptions. For example, in commit `d1f37ff8`, lines 6459-6464 of file `virt/libvirt/driver.py` are not robustly covered because there are two separate blocks of code that raise the same `InvalidNetworkNUMAAffinity` exception with different messages based on the response from the network. The overly-specific flaky test checks for an exact match of one message.

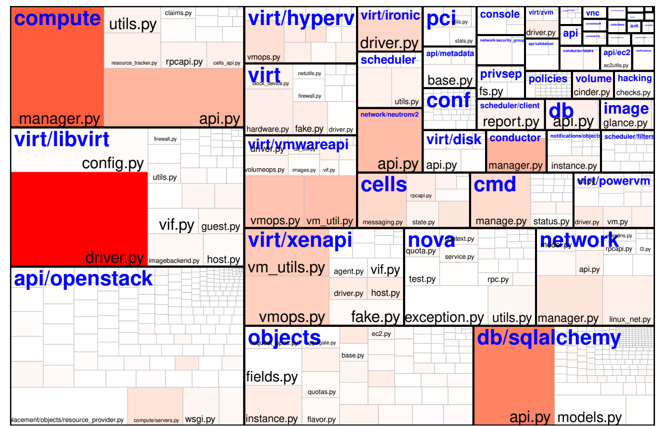
Argument A.2: The flakily covered statements appear in areas of code that are poorly covered in general.

Rationale: Flakily covered statements may be more likely to appear in modules with lax testing practices in general. Since low coverage may indicate that testing is insufficient [6], it may also be an indicator of where flakily covered statements are likely to appear. Such a trivial explanation would threaten the value of our prior observations.

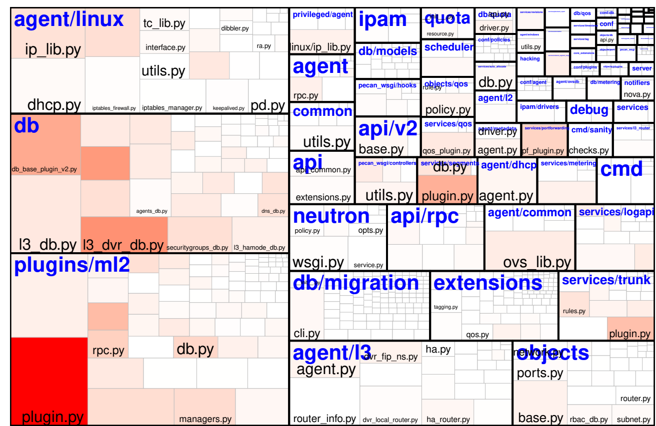
Approach: For each file, we compute the number of uncovered statements and flakily covered statements. Next, we compute the Spearman correlation coefficient (ρ) to measure the strength of the relationship between poor coverage and incidences of flakily covered statements. We choose to use Spearman's ρ rather than Pearson's r because Spearman's ρ can detect non-linear associations. Spearman's ρ ranges from -1 to 1, with 0 indicating no correlation, 1 indicating a positive correlation (i.e., an increase in the incidences of uncovered statements is associated with increases in the incidences of flakily covered statements), and -1 indicating an inverse correlation (i.e., an increase in the incidences of uncovered statements is associated with a decrease in the incidences of flakily covered statements and vice versa). To control for file size, we also compute Spearman's ρ to measure the correlation between the density of uncovered statements and flakily covered statements (i.e., normalized by file size).

Results: For Nova and Neutron, we observe weak ($\rho = 0.36$) and very weak ($\rho = 0.189$) levels of positive correlation between incidences of uncovered and flakily covered statements. While statistically significant, the magnitude of these correlations do not support the AD's hypothesis. Furthermore, in Cinder, we observe a weak level of negative correlation ($\rho = -0.327$), further weakening the argument of the AD.

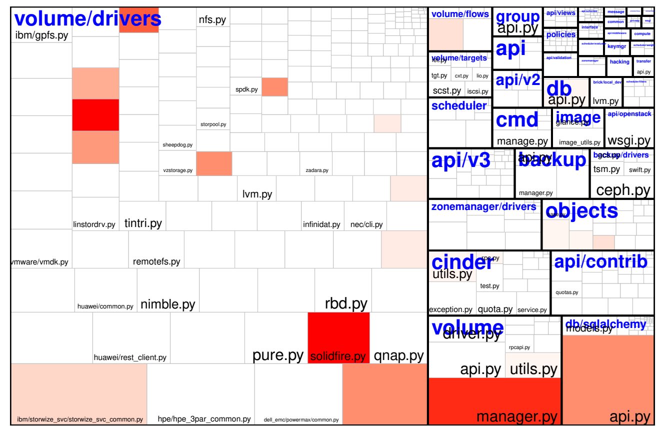
When controlling for file size, for Neutron and Cinder, we observe very weak levels of correlation ($\rho = 0.065$ for Neutron, $\rho = 0.085$ for Cinder). In Nova, we observe



(a) Nova

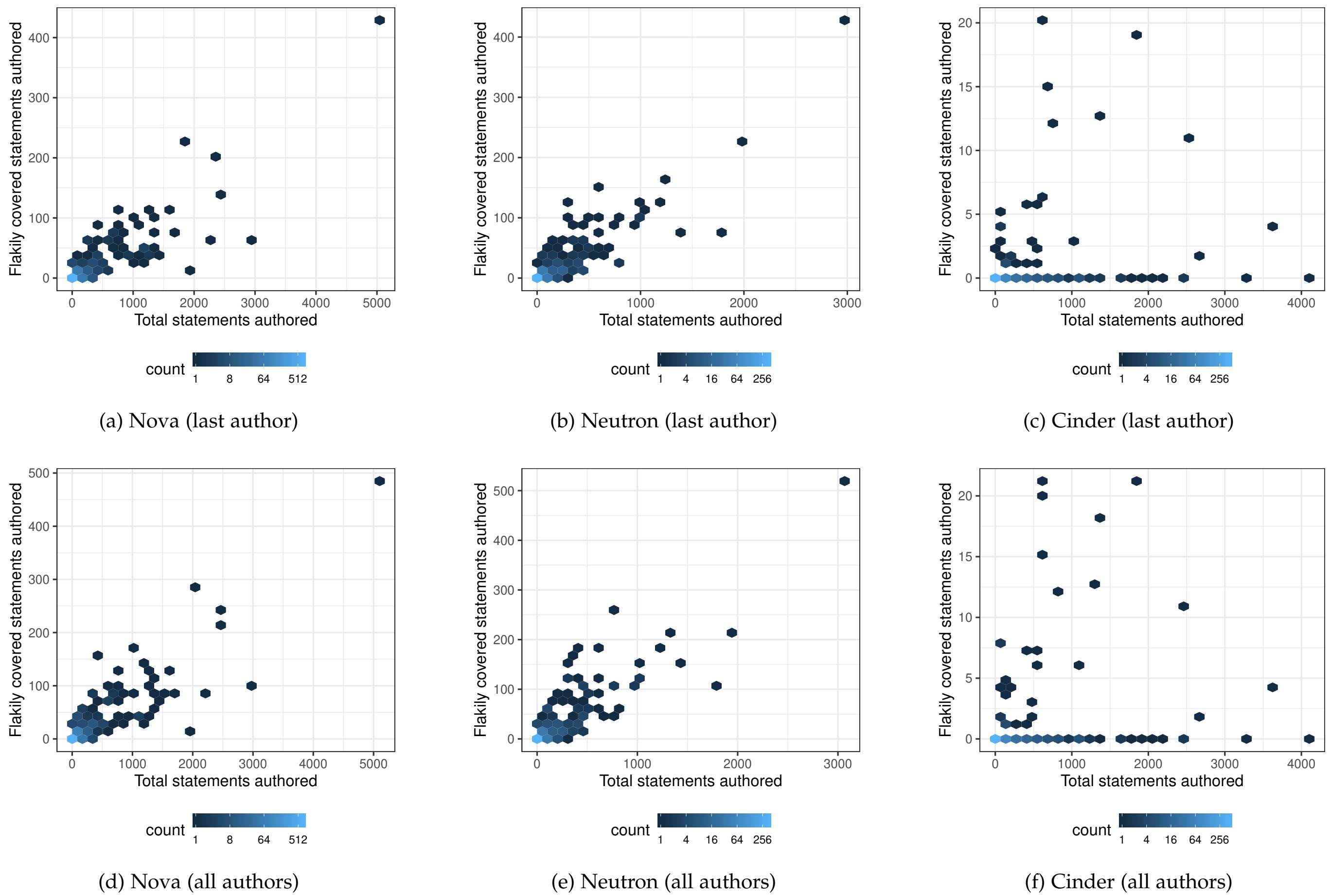


(b) Neutron



(c) Cinder

Fig. 3: Dispersion of flakily covered statements across modules. Although there are hotspots, flakily covered statements are dispersed across modules.



(a) Nova (last author)
(b) Neutron (last author)
(c) Cinder (last author)
(d) Nova (all authors)
(e) Neutron (all authors)
(f) Cinder (all authors)

Fig. 4: For each author, we plot the number of total statements contributed against the number of flakily covered statements contributed. We attribute flakily covered statements to the last author (first row) or all authors (second row) who have modified them.The number of flakily covered statements varies across contributors.

a weak level of negative correlation ($\rho = -0.089$). These density correlation values also do not support the claim that uncovered and flakily covered statements are associated.

Argument A.3: The flakily covered statements are most likely introduced by a small group of developers.

Rationale: Every developer has a characteristic style, ranging from preferences about identifier naming to preferences about object relationships and design patterns. Some styles may result in statements that are hard to robustly cover.

Approach: We use the git blame command to find out the last known author of flakily covered statements. The last known author is a commonly applied heuristic to estimate ownership in previous studies [29]. We also use the git log command to extract the list of authors who have modified flakily covered statements over time. We group the flakily covered statements by author and study their distributions using hexbin plots. A hexbin plot is a variant of a scatter plot where overlapping points are represented by a single hexagonal bin. The shade of the bin indicates the number of points in the bin. For our analysis, we plot the total number of statements authored by a contributor on the X-axis and the number of flakily covered statements authored on the Y-axis.

Results: Figure 4 shows the the number of flakily covered statements varies across contributors. The percentage of contributors who have authored at least one flakily covered statement is 41% for Nova, 46% for Neutron and 5% for Cinder when flakily covered statements are associated with the last author of the statement. When flakily covered statements are associated with all authors, the percentages slightly increase to 43% for Nova and 49% for Neutron, but there is no change for Cinder. In the case of Nova and Neutron, contributors with the highest number of flakily covered statements have also authored more statements in general. On the other hand, in Cinder, the contributor with the most number of lines has not contributed any flakily covered lines.

> **Dispersion**: Flakily covered statements are dispersed across modules and contributors.

B. Ownership

Due to a lack of familiarity, new contributors to a project may not fully comprehend the architecture or design implications of their initial contributions. More experienced contributors would be less likely to make such mistakes.

Project	Statement Last Author Experience (B.1)	Statement All Authors Experience (B.1)	Test Last Author Experience (B.2)	Test All Authors Experience (B.2)	Statement Age (C.1)	Statement Churn (C.2)
Nova	0.0396***	0.04***	0.0535***	0.061***	0.0337***	0.0925***
Neutron	0.1231***	0.026**	NA	NA	0.1156***	**0.2590*****
Cinder	0.0751***	NA	NA	NA	0.06221***	0.0281***

TABLE 1: Comparing flakily covered statements and flaky tests with robustly covered statements and robust tests, respectively. Numbers indicate the Cliff's delta effect sizes, which are negligible unless shown in bold. The asterisks indicate the p-values of the Mann-Whitney U test, where ** indicates $p < 0.01$, and *** indicates $p < 0.001$.

Ownership properties, which are contributor-oriented metrics such as experience, may explain the incidences of flakily covered statements. A naïve explanation of our results may be that flakily covered statements occur because new contributors tend to: (B.1) write statements that result in non-deterministic behaviour or (B.2) write tests that are non-deterministic. Below, we explore these AD arguments.

Argument B.1: New contributors tend to contribute code that is difficult to test robustly.

Rationale: Whenever a block of code is changed, all the tests that cover the block of code must also be verified and updated to reflect changes made to source code. A new contributor who is unfamiliar with the test suite, may be unaware of which tests need to be modified. If the code under test is changed in a way that makes a test flaky, then it will lead to flakily covered statements.

Approach: For each statement, we estimate its author's experience with the project by computing the number of commits that an author has made prior to changing this statement. To conserve space, detailed plots of the distributions have been relegated to the online appendix.[7]

We use Mann Whitney U tests to check whether differences in the distributions are statistically significant. The Mann Whitney U test is a non-parametric test of the null hypothesis that two distributions come from the same population. We adopt a conservative threshold (α=0.01) for rejecting the null hypothesis of our test, which is: *H_0: There is no significant difference between the distributions of author experience of flakily covered statements and robustly covered statements.*

Next, to estimate the practical difference between these distributions, we apply Cliff's delta, a non-parametric effect-size measure. Values of Cliff's delta range between -1 and 1. We adopt the significance levels proposed by prior work [30]: negligible when $0 \leq |\delta| < 0.147$, small when $0.147 \leq |\delta| < 0.330$, medium when $0.330 \leq |\delta| < 0.474$, and large when $0.474 \leq |\delta| \leq 1$. A positive Cliff's delta indicates that values of the first distribution are larger than those of the second distribution, while a negative Cliff's delta indicates the inverse. Similar to Argument A.1, we study the experience of the last author to modify the statement, as well as all authors who have modified the statement.

Results: Column 1 of Table 1 shows the Mann-Whitney U test results of comparing the last-known author experience values. The test results are significant ($p < 0.001$ in all three cases), indicating that we can reject our null hypothesis H_0. However, the effect size is negligible for all three projects, indicating that the practical difference is insignificant.

Column 2 of Table 1 shows the Mann-Whitney U test results of comparing the author experience throughout the history of a statement. For Cinder, the test result is inconclusive ($p > 0.01$). For Nova and Neutron, the test results are significant ($p < 0.001$ for Nova and $p < 0.01$ for Neutron), indicating that we can reject our null hypothesis H_0. However, the effect size is negligible for both projects, indicating that the practical difference is insignificant.

Argument B.2: The flaky tests that lead to flakily covered statements are introduced by new contributors, who lack familiarity with the project.

Rationale: When new contributors write tests, they may not be completely aware of the system runtime conditions. Thus, new contributors may be more prone to writing flaky tests, which in turn will create flakily covered statements.

Approach: We use the same heuristic approach to estimate the experience of authors as we applied in Argument B.1. In this case, we apply the heuristic to test code. We again use Mann Whitney U tests and Cliff's delta effect-size measures to compare distributions statistically, and relegate detailed plots of the distributions to the online appendix.[7]

Results: Column 3 of Table 1 shows the results of comparing the last-known author experience values. For Neutron and Cinder, there is no significant difference in the experience of authors of flaky and robust tests. For Nova, we observe a significant difference ($p < 0.001$); however, the Cliff's delta effect size is negligible.

Column 4 of Table 1 shows the result of comparing author experience values throughout the history of the tests. For Nova, the test results are significant ($p < 0.001$), indicating that we can reject our null hypothesis H_0. However, the effect size is negligible, indicating that the practical difference is insignificant. For Neutron and Cinder the test results are inconclusive ($p > 0.01$).

We cannot conclude that flaky tests are introduced only by new contributors who lack familiarity with the project.

> **Ownership**: The experience of authors of flaky tests and flakily covered statements are often significantly different than the experience of authors of robust tests and robustly covered statements, respectively ($p < 0.001$ in 13 of 18 cases). However, in no case is the difference non-negligible ($\delta < 0.147$), indicating that the difference is of no practical consequence.

7. https://tinyurl.com/flakyprogramelements-appendix

C. Development Activity

In large software systems, different parts of the system change at different rates. The recency and frequency of development activity may already explain where flakiness occurs. Indeed, a naïve explanation of our results may be that the flakily covered statements are: (C.1) are not under active development; or (C.2) undergo plenty of churn. Below, we explore each of these AD arguments.

Argument C.1: The flakily covered statements are statements that are not under active development.

Rationale: Source code is continuously evolving and needs to be actively maintained. However, as software evolves, some areas of the codebase attract more developer attention, while other parts do not. The flakily covered statements that we observe may simply be due to a lack of maintenance priority on the modules where they appear.

Approach: To investigate C.1, we estimate the age of each statement using the number of days since the last change to the statement. We again use Mann Whitney U tests and Cliff's delta effect-size measures to statistically compare the distributions of statement age in robustly and flakily covered statements. Detailed plots of the distributions are available in the online appendix.[7]

Results: Column 5 of Table 1 shows the results of the Mann-Whitney U test, which indicate that the null hypothesis can be rejected, and that there is a statistically significant difference in the age of statements between the two groups. However, the Cliff's delta effect sizes are negligible.

Argument C.2: Flakily covered statements are those that undergo plenty of churn.

Rationale: When statements change, the tests that cover them may also have to change. If test maintenance is neglected, tests may not accurately assess the code under test. Flakily covered statements may be a symptom of the test and production code synchronization problem. In their study, Elbaum et al. concluded that even minor changes in production code can significantly affect test coverage [7].

Approach: To investigate C.2, we compute the amount of churn of each statement, i.e., the number of commits in which the statement has been modified. We again use Mann Whitney U tests and Cliff's delta effect-size measures to statistically compare the churn of flakily and robustly covered statements. Detailed plots of the distributions are available in the online appendix.[7]

Results: Column 6 of Table 1 shows that results of the Mann-Whitney U test, which indicate that there is a significant difference in the rates of churn that flakily and robustly covered statements undergo. However, the Cliff's effect sizes indicate that the practical difference is negligible or small. Therefore, in terms of churn, the flakily covered statements are not considerably different from robustly covered statements.

> **Importance**: Flakily covered statements are similar to robustly covered statements in terms of age and churn.

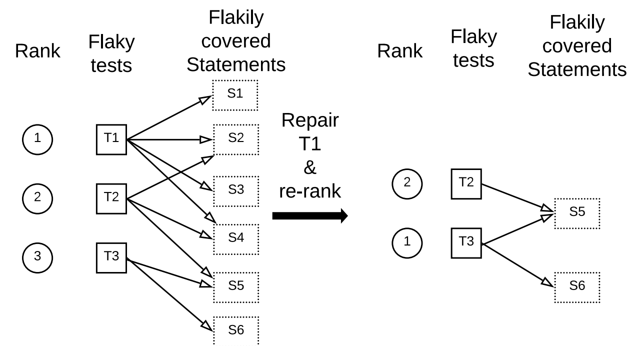


Fig. 5: Illustration of a single iteration of GreedyFlake

6 GreedyFlake: Prioritizing the Repair of Flakily Covered Statements

The prior sections have demonstrated that flakily covered statements are not rare (Section 4) and are not easily explained by basic code, change and contributor characteristics (Section 5). In this section, we shift our focus to prioritizing flakily covered statements for repair, i.e., obtaining robust test coverage of these statements.

Software teams operate with time and budget constraints. Since repairing all of the flakily covered statements would require a substantial investment of time and budget, it is likely impractical to assume that a team can repair all of the flakily covered statements immediately. Software teams would like to prioritize their repair investments such that they will receive the largest return on investment as quickly as possible. Similar to test case prioritization [20], [21], [31], [32], we would like to order (flaky) tests in such a way that the optimal returns are achieved.

Below, we present GreedyFlake—our proposed prioritization approach (6.1), as well as our approach to evaluate GreedyFlake with respect to baseline approaches (6.2) and the evaluation results (6.3).

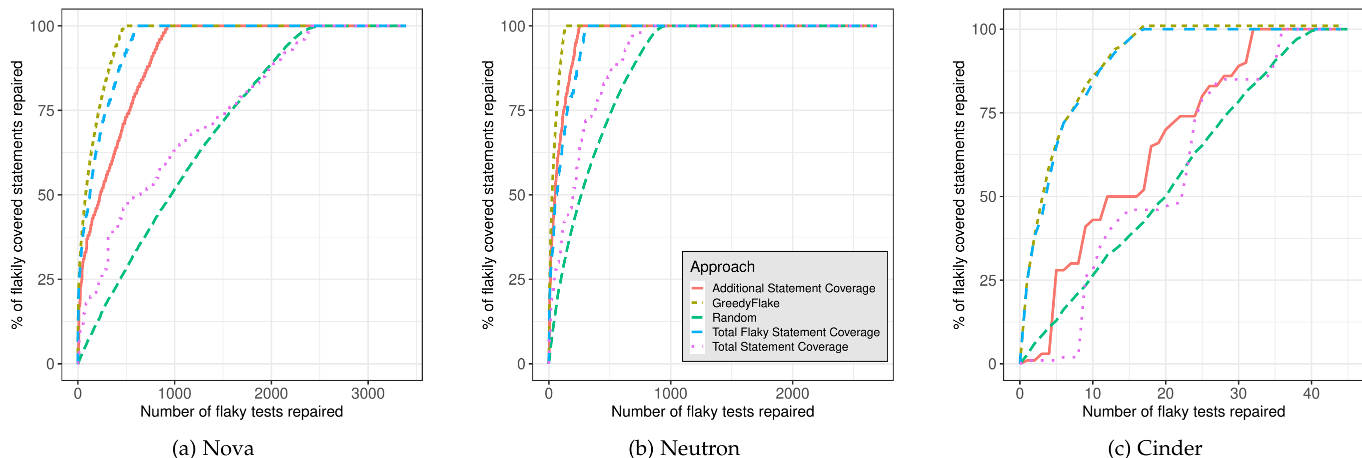
In Section 4, we report the difference in coverage when test characteristics are taken into consideration. In this section, we evaluate the effort that is required to repair flakily covered statements and address the difference in coverage.

6.1 GreedyFlake

GreedyFlake uses a greedy algorithm to order flaky tests for repair. The algorithm consists of ranking and selection steps. In the ranking step, tests are sorted by the number of flakily covered statements that will be repaired if the test is made robust. In the selection step, the top-ranked test from the ranking phase is selected and proposed for repair.

Each repair operation may impact the ranking of which test should be repaired next. For example, in Figure 5, the initial ranking of tests is T1, T2, and T3. Repairing T1 also robustly covers two statements that it shares with T2 (S2 and S4). Since repairing T2 can robustly cover one statement, while repairing T3 can robustly cover 2 statements, in the second step, T3 is ranked above T2.

After each repair recommendation, GreedyFlake performs a re-ranking step. This re-ranking ensures that we



(a) Nova (b) Neutron (c) Cinder

Fig. 6: Comparing various approaches to repairing flakily covered statements. GreedyFlake outperforms random and traditional prioritization approaches.

Project	Random	Total Statement Coverage	Additional Statement Coverage	Total Flaky Statement Coverage	GreedyFlake
Nova	0.68	0.74	0.91	0.95	0.96
Neutron	0.88	0.91	0.97	0.96	0.99
Cinder	0.55	0.57	0.66	0.89	0.90

TABLE 2: GreedyFlake Evaluation: AUCEC of various test case prioritization techniques

select the flaky test that provides the most return on investment at each step. The (re-)ranking and selection processes are repeated until no flakily covered statements remain or until all tests have been suggested for repair.

6.2 Evaluation Setup

In order to evaluate GreedyFlake, we compare GreedyFlake with baseline approaches. The first baseline is a random prioritization approach, where we recommend a randomly selected flaky test for repair at each stage. The random baseline is not selected to be a true baseline, but rather as a sanity check. If our technique underperforms with respect to random guessing, it is truly not worth adopting. We estimate the random baseline empirically by selecting the median performance scores of 100 random orderings.

Previous studies have suggested algorithms for Test Case Prioritization, such as Total Statement Coverage Prioritization (TSCP) and Additional Statement Coverage Prioritization (ASCP). These baselines have been successfully applied to other Test Case Prioritization problems [25], and have been shown to achieve reasonable performance [24]. TSCP sorts tests by the amount of coverage that they provide in descending order. ASCP performs a re-ranking step to select the test that offers the most improvement in coverage. If GreedyFlake underperforms with respect to these baselines, it would be more prudent to prioritize tests based on coverage to repair flakily covered statements, avoiding the costs involved in labelling these statements.

Finally, we compare GreedyFlake with Total Flaky Statement Coverage Prioritization (TFSCP). In TFSCP, we skip the re-ranking step of GreedFlake to determine if re-ranking actually leads to better performance.

We compare the approaches using Alberg diagrams [33]. The GreedyFlake and baseline approaches are each plotted on a grid that shows the cumulative percentage of flakily covered statements that have been repaired (Y axis) against the number of flaky tests that have been repaired (X axis). Lines that climb quicker (i.e., are drawn towards the top-left corner of the grid fastest) are achieving better results.

In addition, for each line, we compute the Area Under the Cost Effectiveness Curve (AUCEC), i.e., the integral of a line in the Alberg diagram space. To do so, we first transform the X axis into a proportion scale, so that both axes of the Alberg diagram range from 0–1. We then compute the AUCEC as $\int_0^1 f(x)dx$, where f(x) is approximated using the collected points in the Alberg diagram space. This AUCEC value ranges between 0 and 1, with 0 indicating the worst performance, 1 indicating the best performance. Our metric AUCEC is similar to the APFD metric Elbaum et al. [21] proposed for evaluating test case prioritization (i.e., the weighted average of the percentage of faults detected).

6.3 Evaluation Results

Figure 6 shows the Alberg diagrams where different approaches are compared. In all of the studied cases, GreedyFlake achieves the top prioritization performance.

Table 2 shows the AUCEC values of each approach. In Nova, GreedyFlake improves over TSCP by 22 percentage points, while improving over ASCP by five percentage points. In Neutron, GreedyFlake still improves over TSCP by eight percentage points. In Neutron, the largest tests tend to be flaky. In Cinder, GreedyFlake improves vastly over random guessing, TSCP, and ASCP. This is because there are only a small number of flakily covered statements, thus

the benefit of an approach that focuses on flakily covered statements is maximized.

When we turn our attention to the improvement achieved by the greedy re-ranking step, we see that re-ranking does not achieve very large improvements. There is a marginal improvement of 1–3 percentage points in AUCEC between GreedyFlake and TFSCP. Nonetheless, the majority of the benefit is achieved by focusing on flakily covered statements, and re-ranking, although reasonable, does not have much of an impact.

> GreedyFlake outperforms random and traditional test case prioritization baseline approaches for prioritizing flaky tests to repair by at least eight percentage points. On the other hand, there is only marginal benefit to the costly re-ranking step (1–3 percentage points), so "greediness" is not necessary to achieve most of the benefit.

7 Threats To Validity

We now discuss the threats to the validity of our study.

7.1 Construct Validity

Construct threats to validity concern the link between theory and real observation.

We categorize a statement as robustly covered if there is at least one robust test covering the statement. In reality, a statement may be considered robustly covered if and only if all the tests covering the statements are robust. Hence, the flaky coverage reported in the study is a lower bound. If statements are more aggressively marked as flakily covered, it will lead to an increase in the number of flakily covered statements and strengthen our claim for the inclusion of reliability in code coverage.

In the evaluation of GreedyFlake, we assume that the cost of repairing any flaky test is equal. However, in reality, some tests are harder to repair than others. The cost of repairing flaky tests depends on many factors, such as the reproducibility of the flakiness, the root cause of the flakiness, the complexity of the test, or the familiarity of the developer with the source code. If a robust measurement for each dimension could be formulated, our prioritization approaches could be re-evaluated as a multi-objective optimization problem. Search-Based Software Engineering (SBSE) approaches could be applied to derive a solution. Nonetheless, in this work, we focus on the prioritization aspect of GreedyFlake, which is a necessary first step.

7.2 Internal Validity

Internal threats to validity concern our ability to rule out other plausible explanations for our results.

We rely on developers to examine test failures and re-run tests to build our corpus of flaky tests. Developers might not always choose to re-run tests or they might not always observe flaky failures. Hence, the flakiness detected through our approach should be interpreted as a lower bound. However, with our approach, we can focus on flakiness that manifests in the continuous integration pipeline and actively tackle flakiness that has concretely impacted development workflows.

Since we did not find strong evidence for the AD arguments, we presume that flakily covered statements are non-trivially explained and would benefit from tool support. It may be that another confounding factor that we have not considered would explain our results. Nevertheless, we analyzed the flakily covered statements from different dimensions of dispersion, ownership and importance. Our observations withstood all three dimensions of confounding factor analysis.

The lower proportion of flakily covered statements results in an imbalanced data set, which can be of concern for statistical inferences. However, the three non-parametric statistical inference techniques applied in this study (Spearman's Rank Correlation, Mann-Whitney U test, and Cliff's delta) do not make assumptions about the distribution of data and are not sensitive to imbalanced data.

7.3 External Validity

External validity concerns have to do with the generalizability of our study. Due to limitations of infrastructure, we were only able to successfully run coverage for three OpenStack projects. However, this study is an exploratory analysis that demonstrates changes in code coverage when test characteristics are considered. We believe that our study could motivate further research in test characteristics-aware code coverage.

8 Conclusion

Code coverage is often used as a quality gate and as a test adequacy metric. Coverage measurements assume that invocation of a program element during any test is equally valuable. Our study explored code coverage with an awareness of test reliability, to further quantify and characterize flakily covered program elements. We also found that flakily covered program elements are not uncommon and their incidences cannot be trivially explained in terms of basic code characteristics.

When prioritizing tests to repair flakily covered statements, we found that our greedy approach produces a more optimal ordering, which differs from existing test effort prioritization approaches. We believe that these observations suggest that developers can (and should) benefit from tool support to manage and mitigate flakily covered program elements.

Replication

To facilitate future work, we have made the data that we collected and the scripts that we used to analyze them available online.[8]

References

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[2] M. Strathern, "'improving ratings': audit in the british university system," *European review*, vol. 5, no. 3, pp. 305–321, 1997.

8. https://github.com/software-rebels/FlakyProgramElements

[3] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[4] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 72–82.

[5] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage measurement experience during function test," in *Proceedings of 1993 15th International Conference on Software Engineering*. IEEE, 1993, pp. 287–301.

[6] B. Marick *et al.*, "How to misuse code coverage," in *Proceedings of the 16th Interational Conference on Testing Computer Software*, 1999, pp. 16–18.

[7] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 2001, p. 170.

[8] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 57–68.

[9] G. Gay, "The fitness function for the job: Search-based generation of test suites that detect real faults," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 345–355.

[10] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[11] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan, "Code coverage and postrelease defects: A large-scale study on open source projects," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1213–1228, 2017.

[12] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron, "Mythical unit test coverage," *IEEE Software*, vol. 35, no. 3, pp. 73–79, 2018.

[13] A. Schwartz, D. Puckett, Y. Meng, and G. Gay, "Investigating faults missed by test suites achieving high code coverage," *Journal of Systems and Software*, vol. 144, pp. 106–120, 2018.

[14] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1. IEEE, 2007, pp. 449–456.

[15] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.

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

[17] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "D e f laker: automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 433–444.

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

[19] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 100–107.

[20] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[21] S. Elbaum, A. G. Malishevsky, and G. Rothermel, *Prioritizing test cases for regression testing*. ACM, 2000, vol. 25, no. 5.

[22] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *15th international symposium on software reliability engineering*. IEEE, 2004, pp. 113–124.

[23] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[24] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[25] E. Shihab, Z. M. Jiang, B. Adams, A. E. Hassan, and R. Bowerman, "Prioritizing the creation of unit tests in legacy software systems," *Software: Practice and Experience*, vol. 41, no. 10, pp. 1027–1048, 2011.

[26] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 112–122.

[27] P. Riehmann, M. Hanfler, and B. Froehlich, "Interactive sankey diagrams," in *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* IEEE, 2005, pp. 233–240.

[28] B. Shneiderman, "Tree visualization with tree-maps: A 2-d space-filling approach," Tech. Rep., 1998.

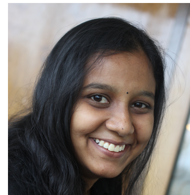
[29] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 491–500.

[30] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

[31] J. J. Li, D. Weiss, and H. Yee, "Code-coverage guided prioritized test generation," *Information and Software Technology*, vol. 48, no. 12, pp. 1187–1198, 2006.

[32] A. Kaur and S. Goyal, "A genetic algorithm for regression test case prioritization using code coverage," *International journal on computer science and engineering*, vol. 3, no. 5, pp. 1839–1847, 2011.

[33] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.



Shivashree Vysali is a Masters student at McGill University, Canada. Her research focuses on understanding and mitigating the impact of flakiness in tests. She received her BSc degree in Computer Science and Engineering from Anna University, India.

Shane McIntosh is an Associate Professor at the University of Waterloo. Previously, he was an Assistant Professor at McGill University, where he held the Canada Research Chair in Software Release Engineering. He received his Ph.D. from Queen's University, for which he was awarded the Governor General's Academic Gold Medal. In his research, Shane uses empirical methods to study software build systems, release engineering, and software quality: http://shanemcintosh.org/.

Bram Adams is an associate professor at Queen's University. He obtained his PhD at the GH-SEL lab at Ghent University (Belgium). His research interests include mining software repositories, software release engineering and the role of human affect in software engineering. His work has been published at premier software engineering venues such as EMSE, TSE, ICSE, FSE, MSR, ASE and ICSME. In addition to co-organizing the RELENG International Workshop on Release Engineering from 2013 to 2015 (and the 1st IEEE Software Special Issue on Release Engineering), he co-organized the SEMLA, PLATE, ACP4IS, MUD and MISS workshops, and the MSR Vision 2020 Summer School. He has been PC co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019.