Enriching Code Coverage With Test Characteristics

Shivashree Vysali Vaidhyam Subramanian

Department of Electrical & Computer Engineering

McGill University, Montréal

August 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Electrical Engineering

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 and only provide a binary covered-or-not classification of program elements. Yet in reality, tests have varied characteristics and coverage can be enriched by incorporating these test characteristics. In this thesis, we expand code coverage classification by adding scope (e.g., unit, function) and reliability (flaky vs. robust) characteristics of the tests to the coverage report. We perform an empirical study of three large software systems from the OpenStack community, namely, Nova, Neutron, and Cinder.

We generate an enriched statement coverage report and glean additional insights. We observe that 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 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. We also 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 order to understand the cost effectiveness of enriching code coverage, 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.

Abrégé

La couverture du code mesure le degré auquel les éléments du code source (par exemple, les instructions, les branches) sont invoqués pendant les tests. Malgré les preuves croissantes que la couverture est une mesure problématique, elle est souvent utilisée pour décider où les efforts de test devraient être investis. Par exemple, en utilisant la couverture comme guide, des tests doivent être écrits pour invoquer les éléments de programme non couverts. À la base, les mesures de couverture supposent que l'invocation d'un élément de programme pendant n'importe quel test est tout aussi précieuse et ne fournit qu'une classification binaire couverte ou non des éléments de programme. Pourtant, en réalité, les tests ont des caractéristiques variées et la couverture peut être enrichie en incorporant ces caractéristiques de test. Dans cette thèse, nous élargissons la classification de la couverture de code en ajoutant des caractéristiques de portée (par exemple, unité, fonction) et de fiabilité (irrégulière vs robuste) des tests au rapport de couverture. Nous effectuons une étude empirique de trois grands systèmes logiciels de la communauté OpenStack, à savoir Nova, Neutron et Cinder.

Nous générons un rapport de couverture des instructions enrichi et recueillons des informations supplémentaires. Nous observons que 60,94% et 63,33% des instructions sont couvertes à la fois par les test unitaires et les tests fonctionnels dans Neutron et Nova, respectivement, alors que seulement 30% sont couvertes par les deux types de tests dans Cinder. Nous constatons que les systèmes sont touchés de manière disproportionnée par les instructions couvertes de manière irrégulière, 5% et 10% des instructions couvertes dans Nova et Neutron étant couvertes de manière irrégulière, respectivement, tandis que ¡1% des instructions Cinder sont couvertes de manière irrégulière. Nous constatons également que les incidences des instructions couvertes de manière irrégulière ne pouvaient pas être bien expliquées en utilisant uniquement les caractéristiques du code, telles que la dispersion, la possession et l'activité de développement. Afin de comprendre la rentabilité de la couverture de code enrichi, nous proposons GreedyFlake - un algorithme de priorisation des efforts de test pour maximiser le retour sur investissement lors de la résolution du problème des éléments de programme couverts de manière irrégulière. Nous constatons que GreedyFlake surpasse les approches de base d'au moins huit points de pourcentage de la zone sous la courbe de rentabilité.

Acknowledgements

This thesis would be incomplete if I do not thank everyone who supported me during the course of this thesis.

First and foremost, I would like to thank my co-supervisors, Dr. Shane McIntosh and Dr. Bram Adams. Thank you for your guidance, motivation and empathy. You are both exemplary researchers and I cherish the opportunity to have worked with you.

I would like to thank Dr. Mei Nagappan, for introducing me to my supervisors and for getting me started on my masters journey. I would also like to thank Dr. Jin Guo, for reviewing this thesis and for her thoughtful recommendations.

I would like to extend my heartfelt gratitude to all the Rebels at the Software Repository Excavation and Build Engineering Labs: Keheliya Gallaba, Christophe Rezk, Farida El Zanaty, and Noam Rabbani. Thank you for the coffee, conversations and camaraderie!

A big shout-out to all my friends, old and new, for the constant support and much needed pep talks. Special thanks to Katy, for the French translation of the abstract.

Finally, I would like to thank family, my parents and my brother, for being receptive

and supportive of all my ideas. Without your encouragement, I would not have been able to start over all the way on the other side of the planet.

My Canadian experience has been truly remarkable (winters included). I am indebted to the True North, for the opportunity to not only be strong and free, but also the privilege to call it my home.

Related Publications

An earlier version of the work in this thesis was published in the IEEE Transactions on Software Engineering:

Quantifying, Characterizing, and Mitigating Flakily Covered Program Elements. Shivashree Vysali, Shane McIntosh and Bram Adams. IEEE Transactions on Software Engineering (TSE), pp. To appear, 2020.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing is one of the most important approaches for assuring the quality of software systems. However, one of the challenges in software testing is evaluating the quality of the test suite itself. Code coverage is a metric that measures the proportion of a program that is tested by a test suite and is often used to assess the quality of test suites. 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 coverage tools 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. Conversely, it is assumed that high coverage indicates adequate testing. For example, the Apache Software Foundation has a quality gate that enforces a minimum code coverage of 80% statements by default.[1] Changes that do not meet this quality criterion are blocked from integration into the product.

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].

1.1 Problem Statement

Code coverage measurements tacitly assume that all tests are of equal value. However, tests have varied characteristics and differ in terms of quality. Developers and test suite maintainers can benefit from understanding these characteristics.

At their core, coverage measurements are based on a coarse-grained binary classification

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

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.

Thesis statement: *Incorporating test characteristics (scope and flakiness) can enrich code coverage and provide additional insights to test suite maintainers.*

In this thesis, we focus on two test characteristics, namely, scope and flakiness, to enrich code coverage. Coverage can be classified based on the scope of the covering test (e.g., unit, functional). Coverage by one scope may not imply coverage by another. It is important that there is coverage by tests of multiple scopes. For example, unit tests can be extensively used to test individual modules and can help pinpoint module-level failures, but they cannot detect failures that are due to module interactions. Functional tests exercise the system in a more realistic setting, but are expensive to execute. Thus, it is challenging to develop a functional test suite that assesses all functionality of the system *thoroughly*. By combining tests of multiple scopes, developers can increase the confidence in the test suite.

Program elements can also be covered by *flaky tests*, i.e., tests that exhibit non-deterministic behaviour. Flaky tests reduce the confidence of developers in the test suite. Program elements that are covered by flaky tests are unlikely to be as well-tested as

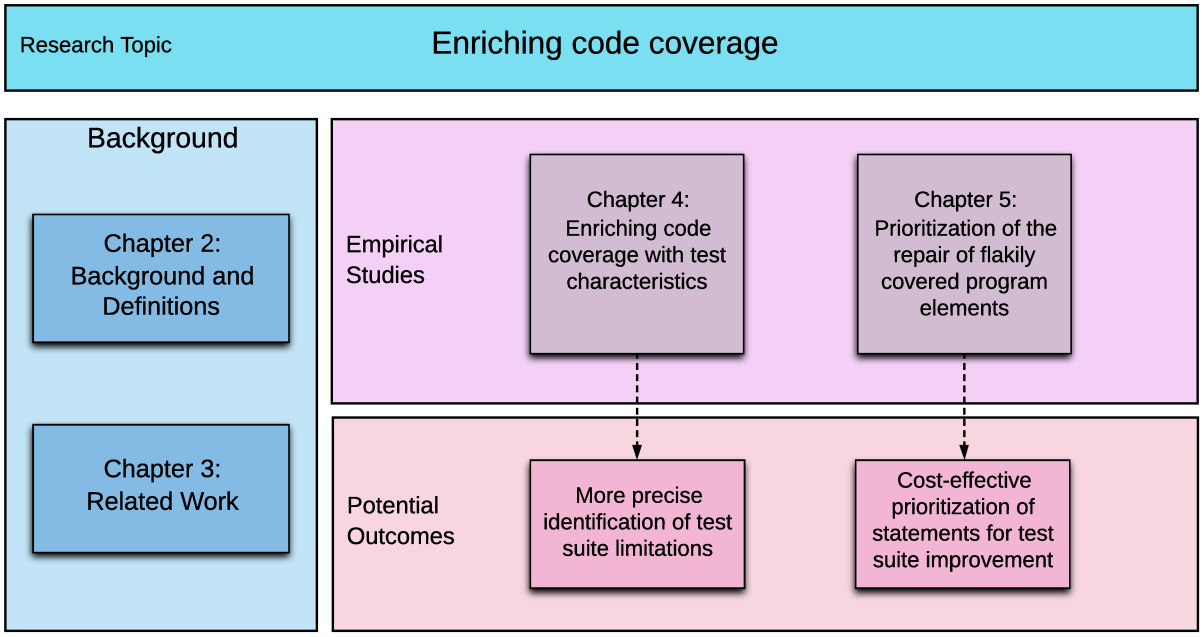


Figure 1.1: An overview of the scope of this thesis.

program elements that are covered by tests with deterministic behaviour.

1.2 Thesis Overview

Figure 1.1 provides an overview of the scope of this thesis. In the first part (blue squares), we provide the necessary background for our topic.

Chapter 2: Background and Definitions

Before discussing how to enrich code coverage, we first provide readers with background information and definitions of terms that we use throughout this thesis.

Chapter 3: Related Work

To situate this thesis with respect to prior research, we present a survey of research on code coverage, flaky tests and test prioritization.

Next, we shift our focus to the main body of the thesis (purple squares in Figure 1.1). In this thesis, we discuss generating an enriched coverage report and leveraging the report for test suite improvement.

Chapter 4: Enriching code coverage

To demonstrate the generation of enriched coverage reports, we conduct an empirical study of three open source projects. We expand the binary code coverage classification by adding *test scope* and *test flakiness*. Then, we take the position of a devil's advocate to further analyze if flakily covered program elements could be attributed to basic code, developer, or maintenance characteristics. If flakily covered program elements can be explained through these basic characteristics, then test suite maintainers can glean insights directly, eliminating the need for an enriched coverage report.

Chapter 5: Prioritization of the repair of flakily covered program elements

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. We propose and evaluate *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.

1.3 Thesis Contributions

The thesis shows that:

- Enriched coverage reports provide insights into test scope and robustness that plain coverage reports may miss. (Chapter 4)
- Flakily covered program elements cannot be explained through basic code characteristics of dispersion, ownership and development activity. (Chapter 4)
- When prioritizing the repair of flakily covered program elements, our proposed approach, GreedyFlake, outperforms multiple test case prioritization approach baselines. (Chapter 5)
- While a targeted approach is beneficial during the prioritization, greediness is not necessary to achieve most of the benefits. (Chapter 5)

1.4 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides background knowledge and definitions of key terms. Chapter 3 surveys previous research related to

code coverage, flaky tests and test prioritization. Chapter 4 describes the data extraction process for enriching code coverage, presents enriched code coverage results and further characterizes flakily covered program elements. Chapter 5 presents GreedyFlake, a prioritization approach to prioritize test cases for repair of flakily covered program elements. Chapter 6 discusses the threats to the validity of this thesis. Finally, Chapter 7 draws conclusion and discusses paths for future work.

Chapter 2

Background

In this chapter, we present the background information and definitions of terms we use throughout the thesis.

2.1 Code coverage

Code coverage is a measurement that is used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests that the program has a lower chance of being defective than a program with low test coverage. To measure what percentage of code has been exercised by a test suite, one or more *coverage criteria* are used. Coverage criteria are usually defined as rules or requirements, which a test suite needs to satisfy.

Commonly adopted coverage criteria include:

- Statement coverage: how many statements from the program have been executed during testing?
- Function coverage: how many functions from the program have been executed during testing?
- Branch coverage: what proportion of branches of each control structure have been executed? For example, given an *if* statement, have tests exercised the condition when it evaluates to true and when it evaluates to false?

In this thesis, we focus on code coverage based on the statement coverage criterion as statement coverage is the simplest coverage criteria and is widely used in practice.

Figure 2.1 shows an example source program (Figure 2.1a), test (Figure 2.1b) and the corresponding statement-level coverage report generated by a code coverage tool (Figure 2.1c). The source program consists of four statements. When the test is run, statements 1-3 are executed and statement 4 is not executed. Statements 1-3 are said to be "covered" by the test. The code coverage in this scenario is $0.75 = \frac{3}{4}$. The report generated by the code coverage tool highlights the covered statements in green and the uncovered statements in red.

2.2 Test characteristics

```python
def isEven(arg):
    if arg % 2 == 0:
        return True
    return False
```

(a) An example program under test

```python
def test_isEven(self):
    result = isEven(2)
    self.assertEqual(result, True)
```

(b) An example test

(c) Coverage report generated by *coverage.py v5.2.1*

Figure 2.1: An example of code coverage. It is to be noted, that these programs are overly verbose for the purposes of the illustration.

Test characteristics refers to various properties of the test(s) that cover program statement(s). Since automated tests are written in code, source code metrics that are used to characterize code can be applied to test code as well. Examples of code metrics include McCabe's cyclomatic complexity [5], Halstead's metrics [6] and design metrics, such as the incidence rate of code smells. In addition to code metrics, there can be characteristics that are unique to test code. For example, when debugging a test failure, the stakeholder might

```
> import sqlite3 as sl
> class Blog:
>   def __init__(self, db):
>     self.conn = sl.connect(db)
>     self.conn.execute(
        '''Create table
        if not exists posts
        (title     TEXT  NOT NULL,
        published INT   NOT NULL)
        ;'''
       )
>      self.conn.commit()

>   def postExists(self, title):
>     q = '''select count(*)
      from posts where title=?;'''
>     c = self.conn.cursor()
>     c.execute(q, [title])
>     res = c.fetchone()
>     r = res[0]
>     if(r > 0):
>       return True
>     return False

>   def create(self, title):
>     if self.postExists(title):
>        print("post exists")
!     else:
!       q = '''insert into posts
         (title, published)
        values (?,0)'''
!       self.conn.
             execute(q, [title])
>     self.conn.commit()
```

(a) source with coverage annotated (> - covered, ! - uncovered)

```
> import unittest
> from unittest.mock import patch
> from blog import Blog

> class TestUnit(unittest.TestCase):
>   def test_post_exists_true(self):
>     with patch('blog.sl') as mocksql:
>       mocksql.connect().
        cursor().fetchone.
        return_value = (1,)
>       b = Blog('mock.db')
>       self.assertTrue(
           b.postExists('title')
        )
>   def test_post_exists_false(self):
>     with patch('blog.sl') as mocksql:
>       mocksql.connect().
        cursor().fetchone.
        return_value = (1,)
>       b = Blog('mock.db')
>       self.assertTrue(
             b.postExists('title')
        )


> class TestFunc(unittest.TestCase):
>   def setUp(self):
>         self.blog = Blog('test.db')

>   def test_create_post(self):
>         self.blog.create('p1')
>         self.assertTrue(
            self.blog.postExists('p1')
          )
```

(b) test

Figure 2.2: A program with multiple tests. Tests `test_posts_exists_true` and `test_posts_exists_false` are unit tests. Test `test_create_post` is a functional test.

wish to know if the test has produced consistent or inconsistent (flaky) outcomes after previous invocations to select an appropriate debugging strategy. The stakeholder might

want to know the scope of the test to determine which parts of the system needs to be examined. Thus, in this thesis, we focus on enriching code coverage with the scope and flakiness characteristics, which we describe below.

2.2.1 The Scope Test Characteristic

Not all tests are written equal. Tests are written with different intended scopes. Unit tests focus on isolating the smallest modules for individual verification. Broadly speaking, because unit tests are so narrow in scope, they are inexpensive to produce and can be executed frequently (e.g., in a continuous integration cycle). Integration or functional tests target logical groups of modules or system-level functionality. Since these tests are broader in scope than unit tests, they tend to be more expensive to produce and execute.

Figure 2.2 shows an example program and associated tests. The program in Figure 2.2a consists of a class Blog, with methods to check if a post exists in against a database. If the does not exist, it is created. Figure 2.2b shows three tests that were written to verify the program. `test_posts_exists_true` and `test_posts_exists_false` are unit tests that were written to test the `postExists` method. These unit tests verify the logic of the `postExists` method without invoking any other dependencies. The external database dependency has been mocked to isolate the method under test.

`test_create_post` is a functional test, as it tests an expected behaviour of the entire system. Unlike the unit tests, these functional tests do not mock the database. The `create`

method is covered only by a functional test. The `postExists` method is covered by both unit and functional tests.

We conjecture that areas of a program that are covered by tests of varying scopes have been more comprehensively verified than areas of code that have been tested in one scope.

2.2.2 The Flakiness Test Characteristic

To aid in debugging, tests should produce repeatable, deterministic results. If the program under test has not changed, the test outcome should not change. However, tests can exhibit non-deterministic behavior. For example, asynchronous waits, network operations, randomly ordered collections, and concurrency can create opportunities for non-deterministic test outcomes [7]. The outcomes from these so-called flaky tests are difficult to diagnose and act upon.

In Figure 2.3b, `test_get_posts` is flaky. The flakiness is due to the assertion in the test, which assumes that the order of elements in the list being returned by the `getPosts` method does not change. However, the order of elements in a list data structure is not guaranteed. Hence, the assertion can sporadically fail with an error as shown in Figure 2.3c.

Flaky tests may indicate the presence underlying issues with the source or test code and reduce stakeholder confidence in the test suite. Program elements that are only covered only by flaky tests are unlikely to be as comprehensively verified as program elements that are covered by robust tests with deterministic behaviour. In the context of this thesis, we define

```
1  import sqlite3 as sl
2
3  class Blog:
4  ...
5    def getPosts(self):
6      posts = []
7      q = '''
8          select title from posts;
9      '''
10     c = self.conn.cursor()
11     res = c.execute(q)
12     for row in res:
13         posts.append(row[0])
14     return posts
```

(a) source

```
1  import unittest
2  from blog import Blog
3
4  class TestBlogFunc(unittest.TestCase):
5    def setUp(self):
6      self.blog = Blog('test1.db')
7
8    def test_get_posts(self):
9      self.blog.create('post2')
10     self.blog.create('post3')
11     p = self.blog.getPosts()
12     self.assertEqual(p,
13         ["post2", "post3"]
14     )
```

(b) test

```
1   ======================================================================
2   FAIL: test_get_posts (test_blog.TestBlogFunc)
3   ----------------------------------------------------------------------
4   Traceback (most recent call last):
5   File "test_blog.py", line 12, in test_get_published
6       ["post2","post3"] )
7   AssertionError: Lists differ: ['post2','post3'] != ['post3','post2']
8
9   First differing element 0:
10  'post2'
11  'post3'
12
13  - ['post2', 'post3']
14  + ['post3', 'post2']
15
16  ----------------------------------------------------------------------
```

(c) Assertion fails due to random ordering of elements in the returned list `posts`

.

Figure 2.3: Source program and test with a flaky failure.

a statement as "flakily covered" if all of the tests covering that statement are flaky.

2.3 Test case prioritization

Large codebases require large test suites. Since executing tests in an arbitrary order may yield suboptimal results (e.g., slow integration feedback for stakeholders), software teams will need to prioritize test execution.

Rothermel et al. [8] formally defined the test case prioritization (TCP) problem as follows:

The Test Case Prioritization Problem:

Given: T , a test suite; PT , the set of permutations of T; f , a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geqslant f(T'')]$.

In this definition, PT represents the set of all possible orderings of T, and f is a function that, when applied to an ordering, yields a fitness value for that ordering. Test case prioritization is performed to achieve a (set of)goal(s), which are qualitatively stated. To measure the success of the prioritization technique, the goal must be quantitatively described; f represents such a quantification.

In the context of this thesis, our qualitative goal is to improve the reliability of a test suite by mitigating flakily covered program elements. We strive to prioritize the repair of flaky tests to achieve this goal as quickly as possible. We measure the success of the prioritization technique by studying Alberg diagrams, which plot the cumulative percentage of flakily

covered program elements that can be mitigated (Y axis) against the number of flaky tests that are chosen for repair (X axis). In our context, a prioritization technique is successful if its curve on the Alberg diagram climbs more quickly, meaning that it maximizes the repair of flakily covered elements and minimizes the number of flaky tests to be repaired.

Chapter 3

Related Work

In this chapter, we present the related work with respect to code coverage, test flakiness and test prioritization.

3.1 Code Coverage

Code coverage is a well established concept in software engineering research and practice. Piwowarski et al. [9] explained that IBM used code coverage in the late 1960s. Marick [10] warns that "requiring" high coverage might lead to tests being written only to satisfy coverage conditions and not to reveal bugs. Elbaum et al. [11] 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 measurements.

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, 12, 13] while other studies do not [3, 14–16]. Schwartz et al. [17] 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. [18] 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 scope and flakiness.

3.2 Flaky Tests

Previous studies on flaky tests have focused on understanding the root causes of flaky tests. Luo et al. [7] 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. [19] performed a similar analysis for Android applications and reported three other root causes, i.e., Dependency, Program Logic,

and UI.

The common practice to determine if a test is flaky is to repeatedly execute a test under scrutiny several times. Tests with inconsistent outcomes in this scenario are labelled as flaky. Since repeating tests is expensive, many studies have focused on improving the detection of flaky tests. Bell et al. [20] proposed *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. [21] proposed an automated approach to detect order-dependent flaky tests. King et al. [22] proposed an approach that leverages Bayesian networks for classifying flaky tests.

In this thesis, we rely on test execution history from continuous integration pipelines to build a corpus of flaky tests. We then build and trace a test-to-statement map to identify flakily covered program elements.

3.3 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. [8] 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. [23] extended the empirical study of Rothermel et

al. by including more programs and prioritization techniques. Do and Rothermel [24] applied coverage-based prioritization techniques to the JUnit testing environment and showed that prioritized execution of JUnit test cases improved the fault-detection rate, i.e., test case prioritization was successfully used to achieve the target objective of improving the fault-detection rate.

In this thesis, we propose a greedy approach to order flaky tests by the number of flakily covered statements for which they are solely responsible. Greedy algorithms have also been explored in prior software testing research. For example, Jones and Harrold [25] proposed a greedy variant to the Modified Condition/Decision Coverage (MC/DC) criterion for prioritization. Moreover, Li et al. [26] 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. Inspired by their promising results, we propose GreedyFlake (Chapter 5) 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. [27] leveraged the development history of a project to generate a prioritized list of functions on which unit testing resources should focus. In our work, we obtain a prioritized list of flaky tests to minimize flakily covered program elements.

Chapter 4

Enriching code coverage

In this chapter, we present our empirical study to enrich code coverage. The goal of this study is to quantify how coverage varies when analyzed with an awareness of test characteristics (scope and flakiness) and to determine if the insights gleaned from the enriched coverage report can be obtained using basic code characteristics, such as, dispersion, ownership and development activity.

We set out to analyze code coverage with an awareness of the scope and flakiness characteristics of the test(s) that cover(s) each statement in the source code. To achieve our goal, we conduct an empirical study of three projects from the OpenStack open-source community, i.e., Nova, Neutron and Cinder, for which we generate the enriched code coverage report by categorizing covered statements based on scope and flakiness. We quantify the difference in code coverage when test characteristics are considered.

Then, we focus on statements that are covered only by flaky tests, which we refer to as “flakily covered statements”. We take the position of a devil's advocate and argue that the occurrences of flakily covered program elements can be explained through either code dispersion, ownership or developer characteristics. The goal of this analysis is to evaluate whether the insights about flakily covered statements from the enriched coverage report can be trivially gleaned through basic code characteristics.

4.1 Study Design

In this section, we outline the design of our approach to empirically study coverage with an awareness of test scope and flakiness.

4.1.1 Studied Systems

In order to analyze code coverage with an awareness of test scope and flakiness, we need data from a software community that follows 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 also need the studied data to be extracted from large and actively developed OpenStack Projects to maximize our chances of observing flaky tests. We start by identifying projects that form the core of OpenStack. According to OpenStack

documentation,[1] Nova, Neutron, Cinder, Keystone, Glance, Swift and Horizon are the core projects.

Next, we need to ensure that we are able to collect complete coverage information by executing the test suites in our instrumented environment. Most OpenStack core projects use Tox to install the dependencies needed for testing.[2] Using this Tox environment, we could successfully replicate the testing environments for Nova, Neutron and Cinder. For Keystone, Glance, Swift, and Horizon, we were unable to replicate the testing environment after a non-trivial amount of effort. The test suites for these projects required setting up multiple external dependencies and were compute intensive. For example, in the case of Keystone, the functional tests required an in-memory key-value store.[3]

4.1.2 Data Extraction

Figure 4.1 provides an overview of the four steps involved in the coverage and test characteristics data extraction process. We explain each step below.

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

[1] https://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html
[2] https://tox.readthedocs.io/en/latest/
[3] https://docs.openstack.org/keystone/latest/contributor/testing-keystone.html

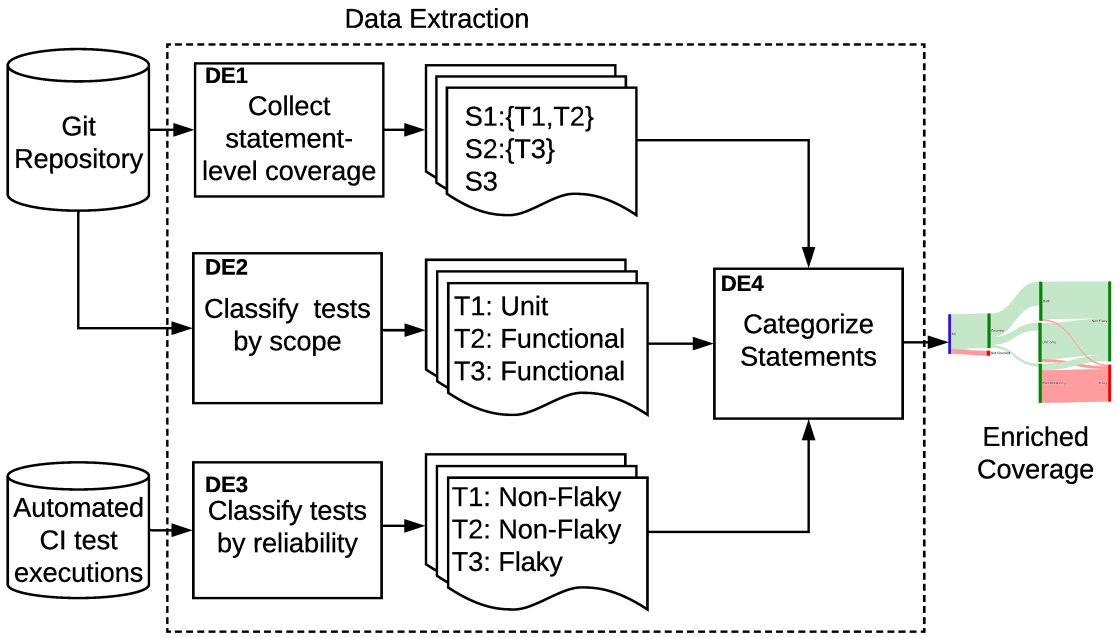


Figure 4.1: An overview of our data extraction approach

analysis. Since we set out to analyze scope- and flakiness-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,[4] 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. [28] demonstrated that flaky tests can yield unreliable coverage measurements. To mitigate the risks posed by flaky tests, for each project, we repeat the

[4]https://coverage.readthedocs.io/en/coverage-5.1/

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 a project-sensitive phenomenon.

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.[5] 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 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. If there were instances of shared code at the test class level, our approach could be easily extended to add this shared code to the test-to-statement mapping by appending it to all of the tests in the test class in which it appears (as is done for setUp/tearDown methods).

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

[5]https://docs.python.org/3/library/unittest.html

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: Classify tests by reliability

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 stakeholders as much as statements covered by robust tests.

Previous studies [7, 20] 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 at least one inconsistent test outcome. 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 flakiness of the tests covering the statement. If all of the tests covering a statement are flaky, the statement is

considered *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*.

4.2 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 flakiness. We visualize the coverage split using a Sankey diagram [29]. Sankey diagrams are variants of flow diagrams, in which the width of arrows is proportional to flow quantity.

Figure 4.2 shows the three-tiered Sankey diagram generated for Nova. 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 flakiness (flakily-covered, non-flakily covered). Figure 4.3 and Figure 4.4 are the Sankey diagrams for Neutron and Cinder, respectively.

The Sankey diagrams show 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, 63.33% and 60.94% of statements are covered by both unit and functional tests in Nova (Figure 4.2)

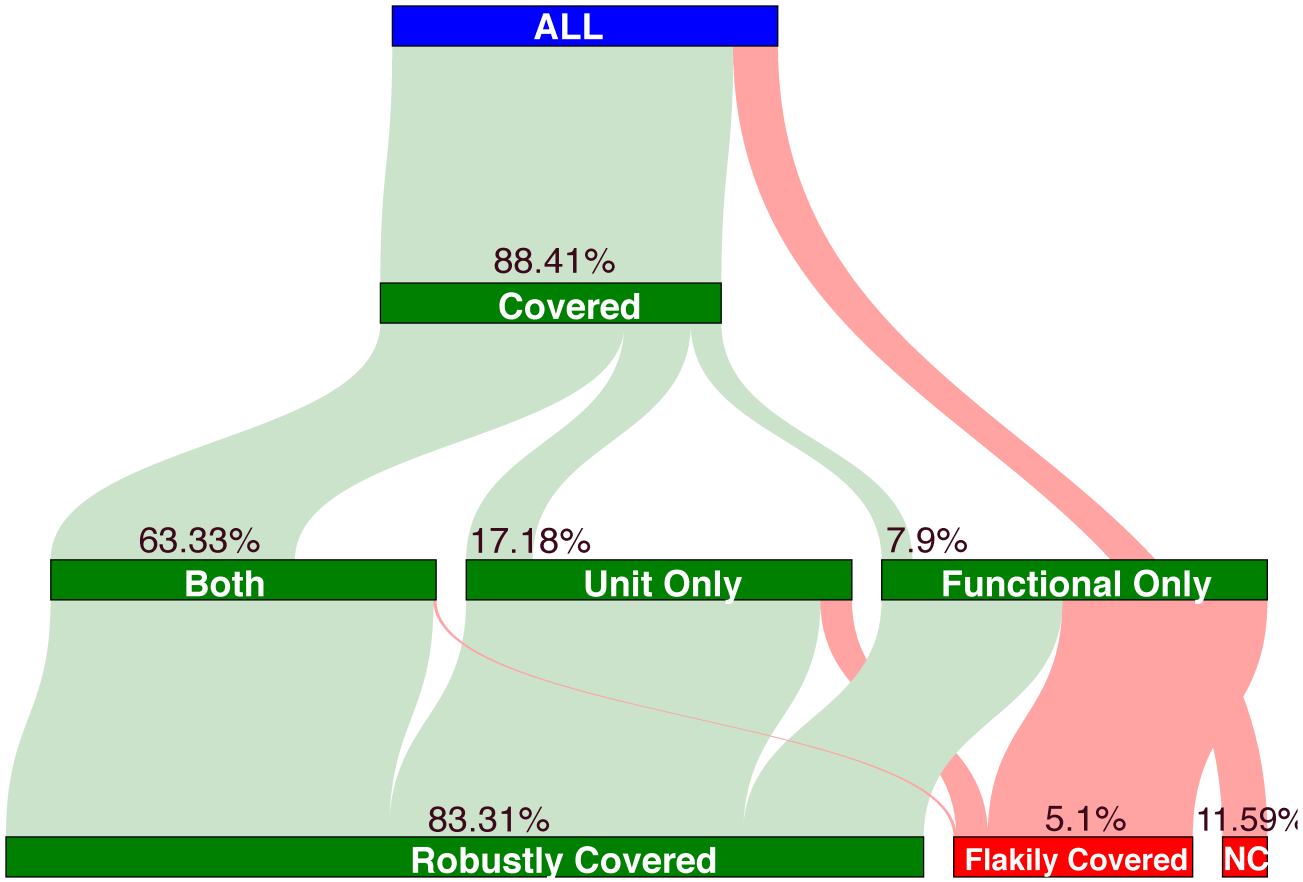


Figure 4.2: Sankey diagram for Nova

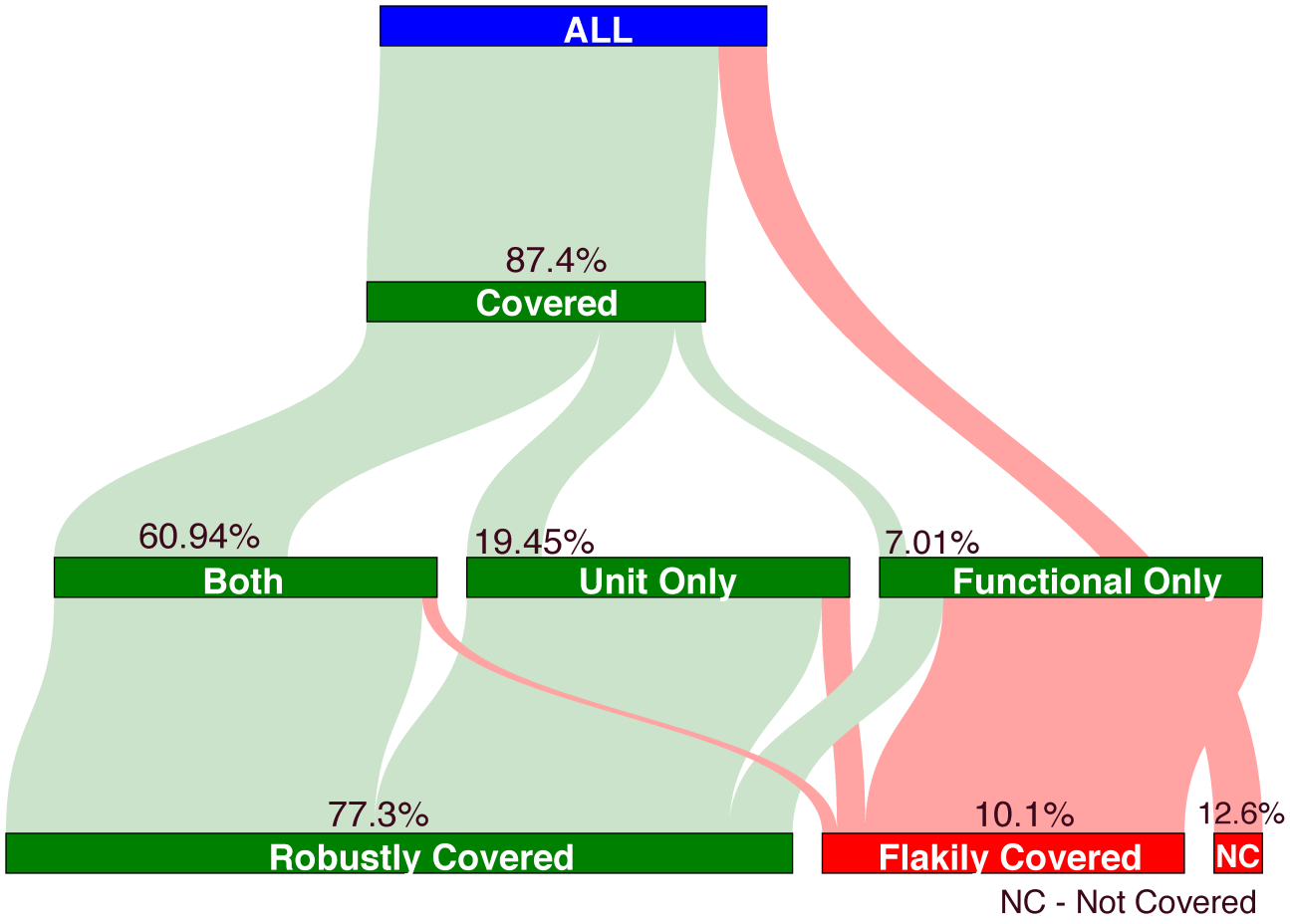


Figure 4.3: Sankey diagram for Neutron

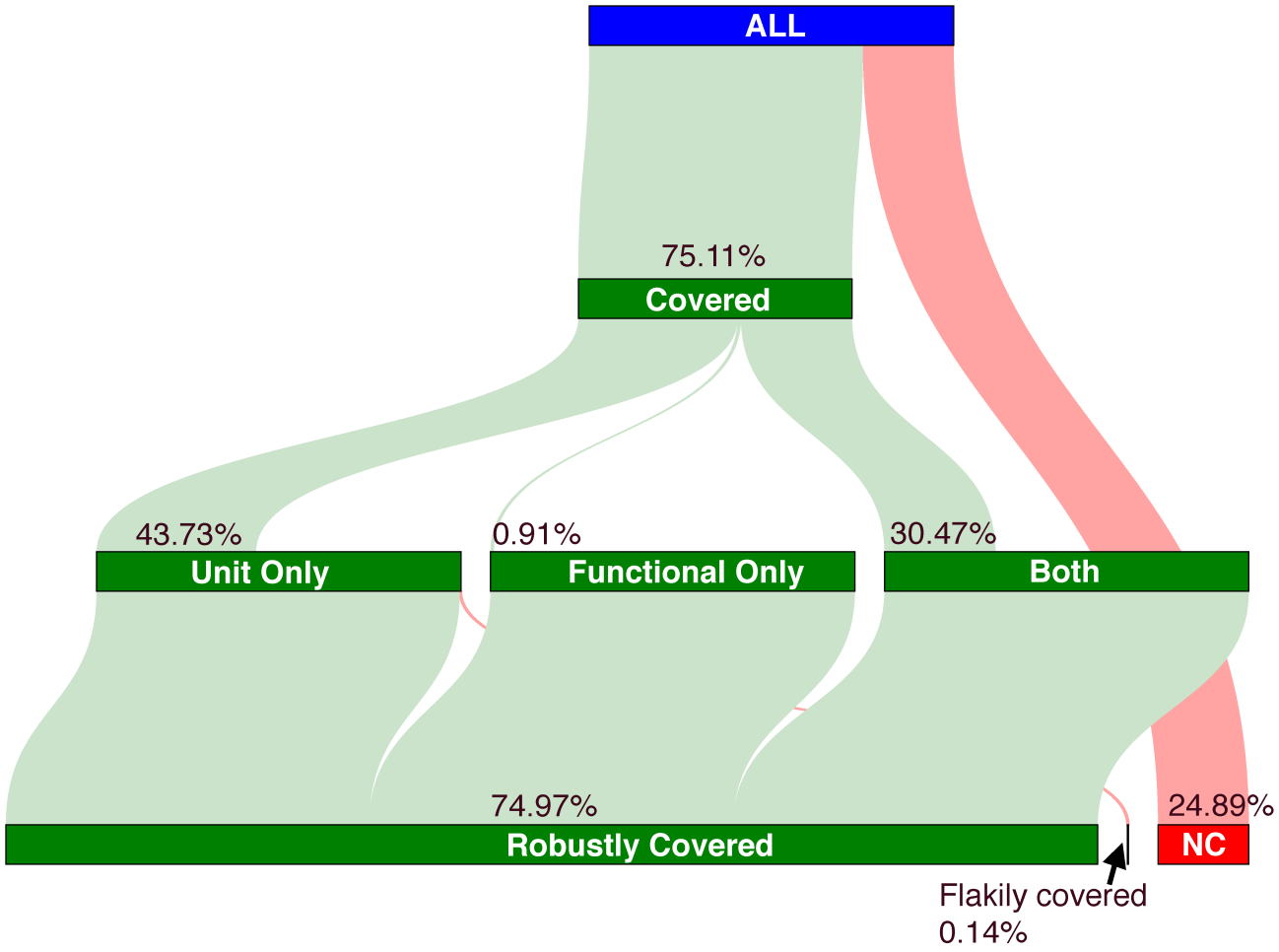


Figure 4.4: Sankey diagram for Cinder

and Neutron (Figure 4.3), respectively, while only 30% are covered by both types of tests in Cinder (Figure 4.4). We suspect this discrepancy is caused by the lower proportion of functional tests in Cinder (5%).

Figure 4.4 also shows that Cinder has the lowest coverage at 75.11%, while, Nova and Neutron have a coverage of 88.41% and 87.4% respectively. 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 become comparable. This further supports the claim that higher coverage scores do not always indicate more thorough testing [10].

The Sankey diagrams help stakeholders to identify possibly weakly covered statements. For example, the Sankey diagram for Neutron (Figure 4.3) shows 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 addressing 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%).

4.3 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 identified 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

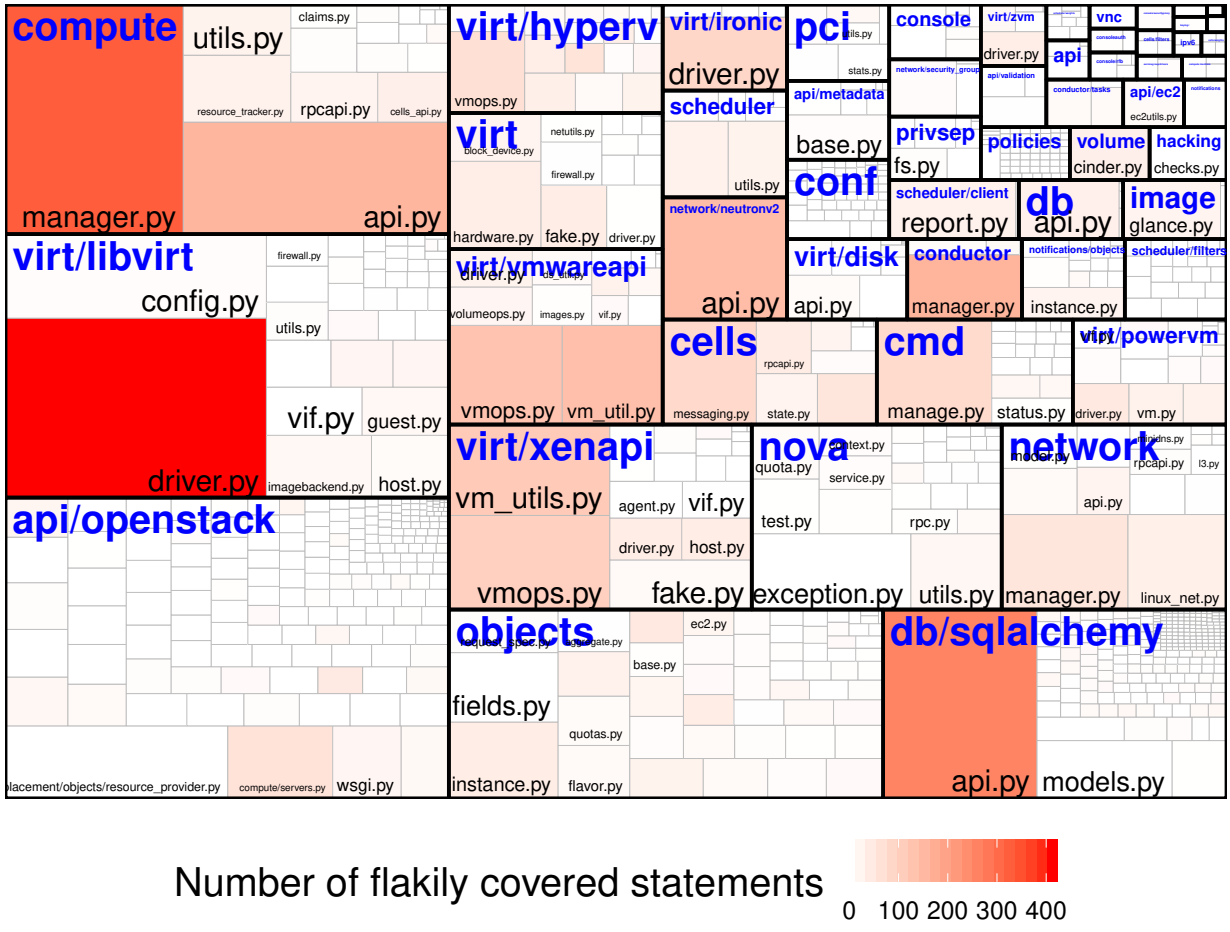


Figure 4.5: Dispersion of flakily covered statements across modules for Nova

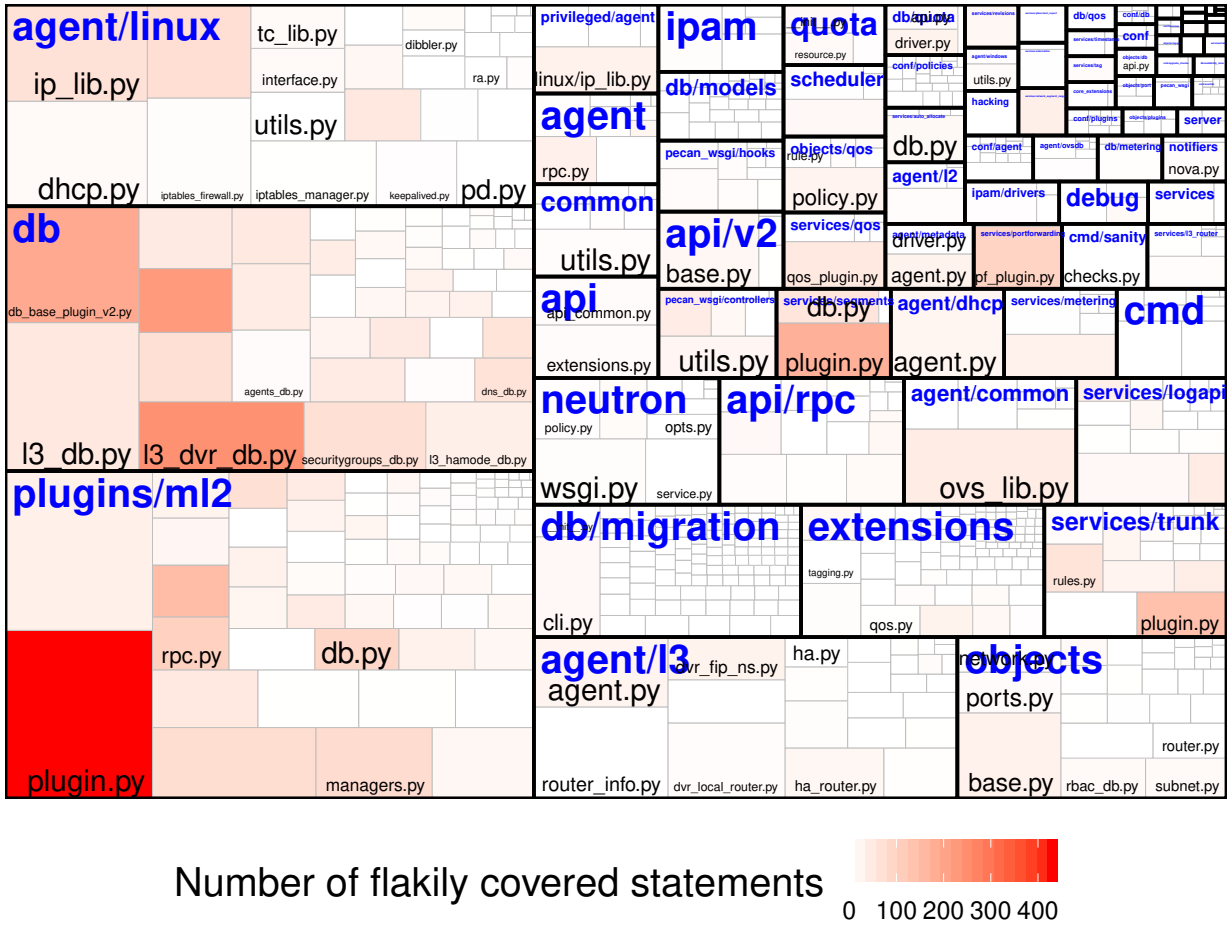


Figure 4.6: Dispersion of flakily covered statements across modules for Neutron

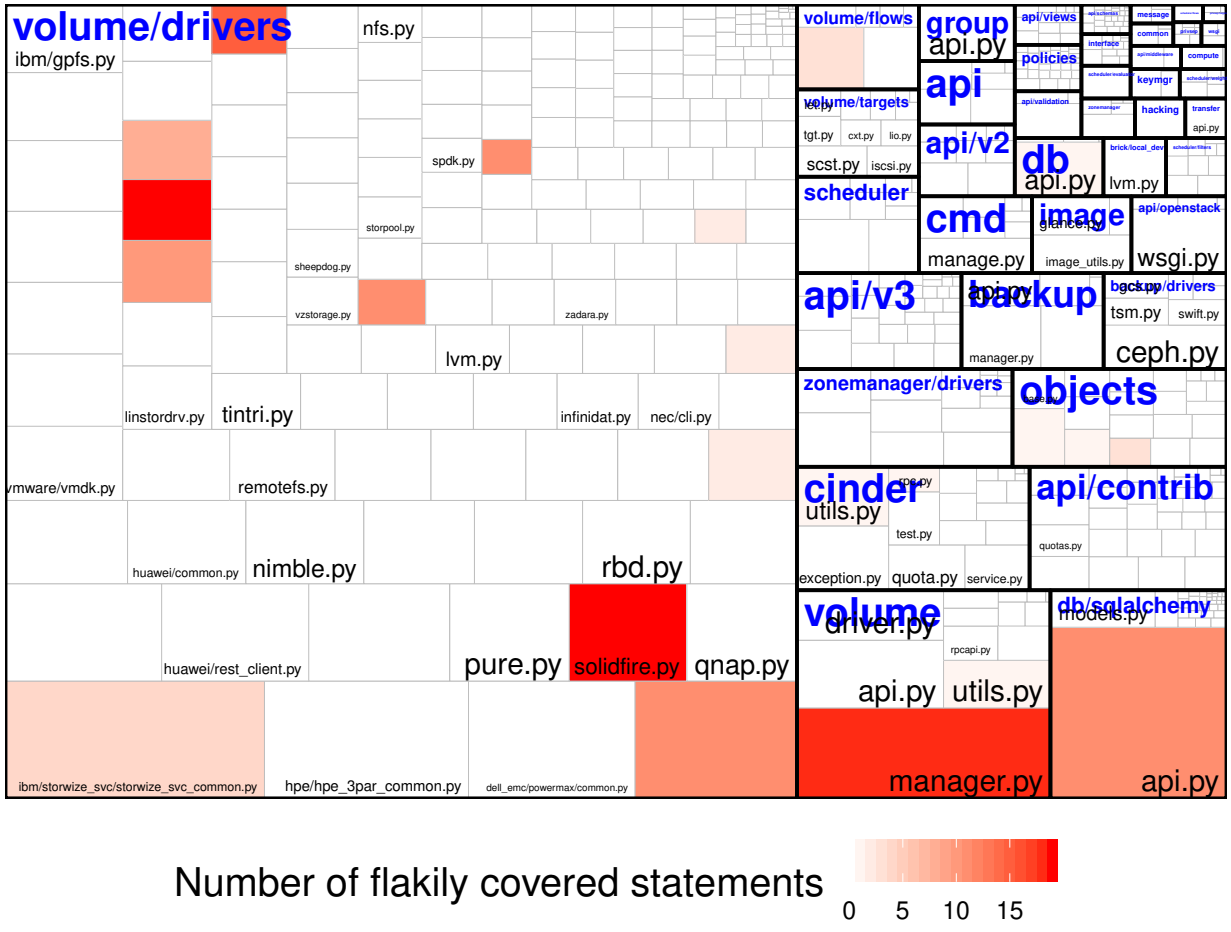


Figure 4.7: Dispersion of flakily covered statements across modules for Cinder

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 [30] 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 that it contains (darker shaded files indicate more flakily covered statements).

Results: We observe that flakily covered statements are often dispersed across modules. Figures 4.5, 4.6, and 4.7 shows the dispersion of flakily covered statements across modules for Nova, Neutron, and Cinder, respectively. 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 4.7 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.

We also observe that while the module-level dispersion of flakily covered statements is often quite high, 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. Figure 4.5 shows that `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. [7] 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 [10], 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

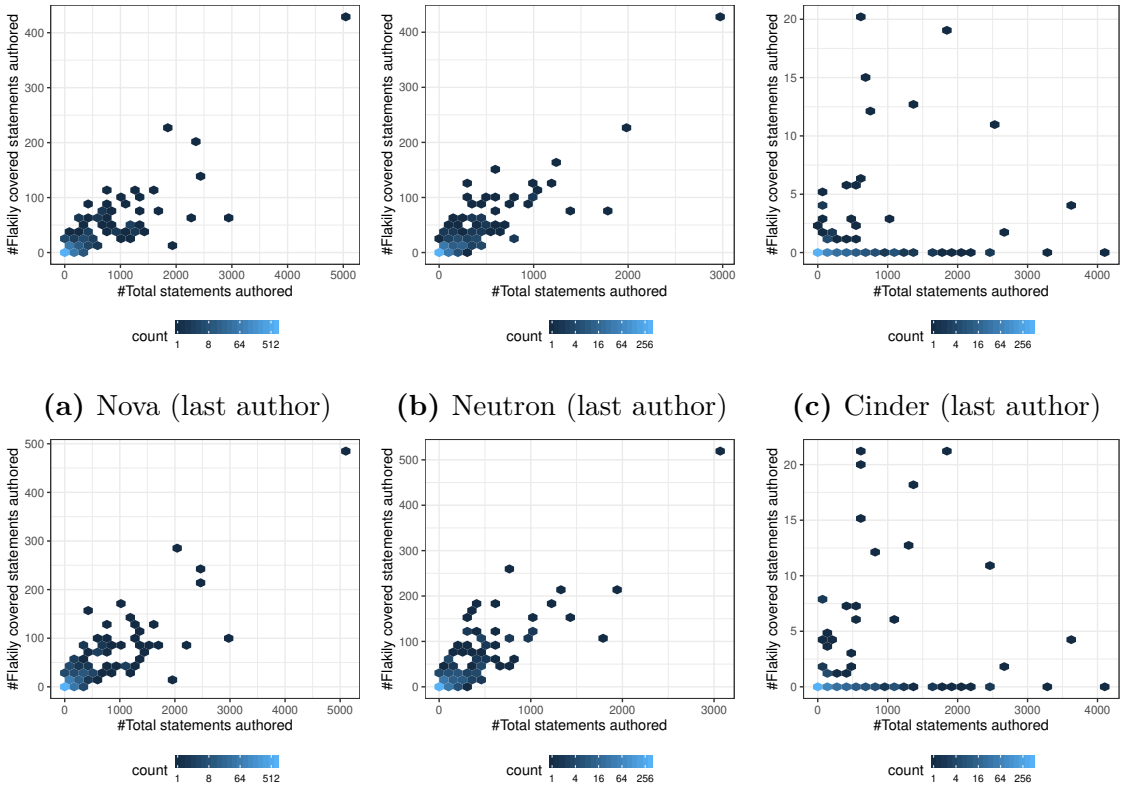
hypothesis. Furthermore, in Cinder, we observe a weak level of negative correlation ($\rho = -0.327$), further weakening the argument of the AD.

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 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 [31]. 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



(a) Nova (last author) **(b)** Neutron (last author) **(c)** Cinder (last author)

(d) Nova (all authors) **(e)** Neutron (all authors) **(f)** Cinder (all authors)

Figure 4.8: 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.

flakily covered statements authored on the Y-axis.

Results: Figure 4.8 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, rather than only the last author to modify the statement (the git log instead of git blame approach), 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 of the largest number of statements has not contributed any flakily covered statements.

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. 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. We use beanplots to compare the distributions of author experience of flakily covered statements to that of robustly covered statements. The detailed plots of the distributions are available in the appendix of this thesis. (Appendix A)

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 could be sampled 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 measurement. Values of Cliff's delta range between -1 and 1. We adopt the significance levels proposed by prior work [32]: 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 4.1 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 4.1 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

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 4.1: 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 $p < 0.001$. "NA" indicates that are is no significant difference between the compared distributions.

Mann Whitney U tests and Cliff's delta effect-size measurements to compare distributions statistically and relegate detailed plots of the distributions to Appendix A.

Results: Column 3 of Table 4.1 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 4.1 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 ($p > 0.01$).

In summary, our quantitative data does not support the conclusion 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.

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 measurements to statistically compare the distributions of statement age in robustly and flakily covered statements. Detailed plots of the distributions are available in Appendix A.

Results: Column 5 of Table 4.1 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. Elbaum et al. [11] observed that even minor changes in production code can significantly affect test coverage.

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 measurements to statistically compare the churn of flakily and robustly covered statements. Detailed plots of the distributions are available

in Appendix A.

Results: Column 6 of Table 4.1 shows the 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.

4.4 Chapter Summary

In this chapter, we present enriched coverage reports for the projects studied and derive additional insights. 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%). We also 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 order to determine whether the insights gleaned from the enriched coverage report can be explained by basic code characteristics, we take the position of a devil's advocate and analyze flakily covered statements along dispersion, ownership and development activity

dimensions of code characteristics. From our analyses, we conclude that the occurrence of flakily covered statements cannot be well explained solely by these basic code characteristics.

In the next chapter, we shift our focus to prioritizing flakily covered statements for repair, i.e., obtaining robust test coverage of flakily covered statements.

Chapter 5

Prioritization of the repair of flakily covered program elements

In chapter 4, we demonstrate that flakily covered statements are not rare (Section 4.2) and are not easily explained by basic code, change and contributor characteristics (Section 4.3). Hence, In this chapter, 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 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

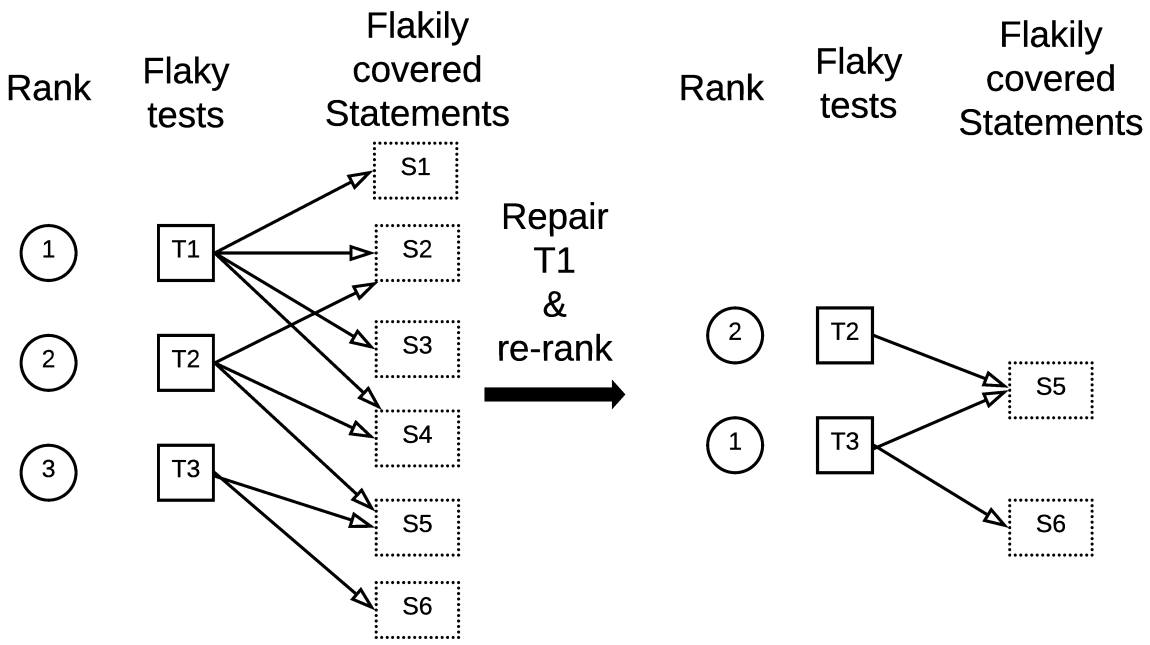


Figure 5.1: Illustration of a single iteration of GreedyFlake

case prioritization [8, 23, 33, 34], we would like to order (flaky) tests in such a way that the optimal returns on investment are achieved.

Below, we present GreedyFlake—our proposed prioritization approach (Section 5.1), as well as our approach to evaluate GreedyFlake with respect to baseline approaches (Section 5.2) and the evaluation results (Section 5.3).

5.1 GreedyFlake

GreedyFlake uses a greedy algorithm to order flaky tests for repair. Similar to other test prioritization approaches [8], [26], we make the simplifying assumption that the fixes for all flaky tests are equally difficult. 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 5.1, 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 two statements, in the second step, T3 is ranked above T2.

Once a flaky test is recommended for repair, the statements covered by this test are removed from the set of flakily covered statements to simulate the repair of the flaky test. Then, GreedyFlake performs a re-ranking step. This re-ranking ensures that we 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.

5.2 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 [27], and have been shown to achieve reasonable performance [26]. 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 [35]. The GreedyFlake and baseline

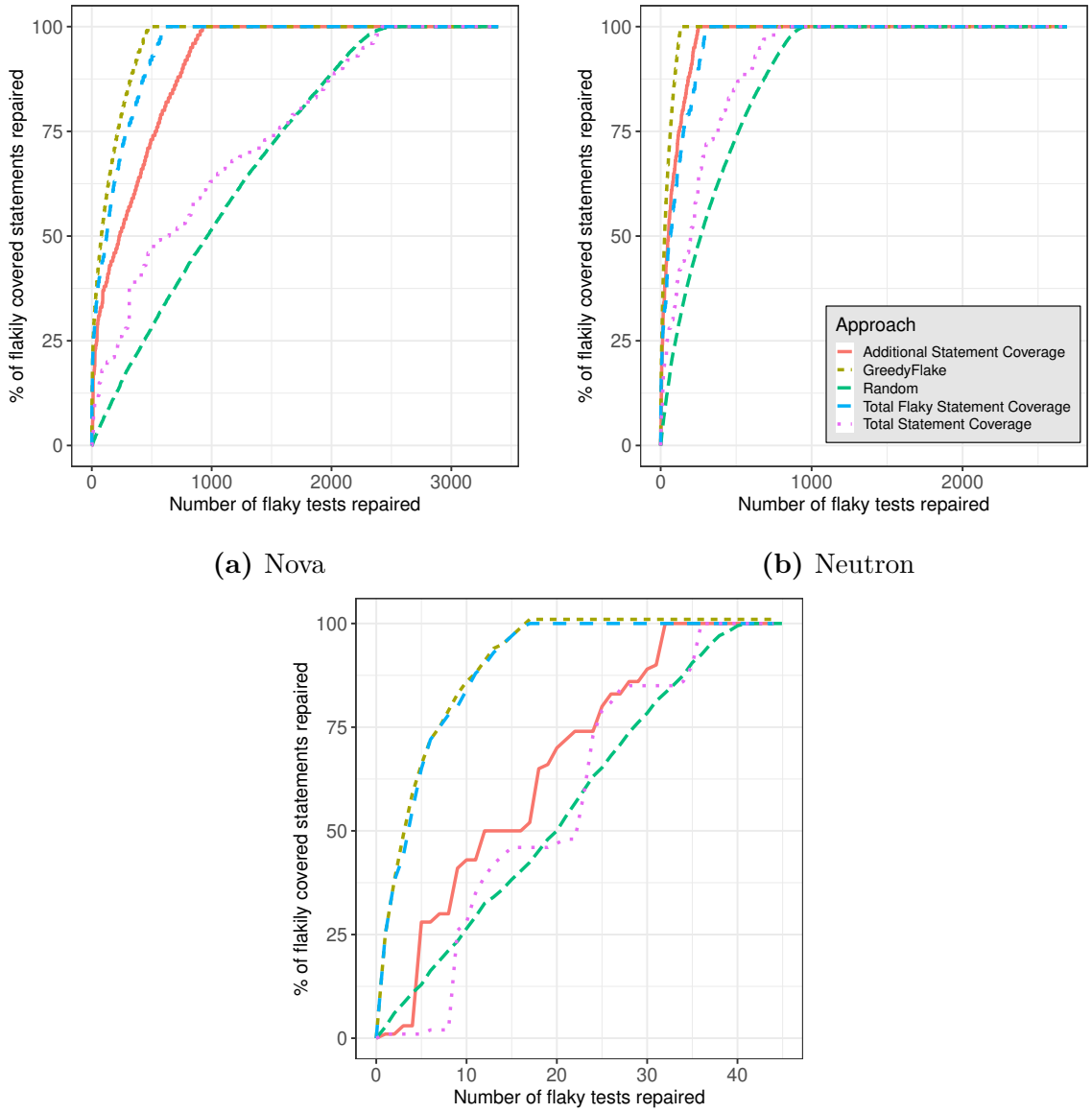
approaches are each plotted on a grid that shows the cumulative percentage of flakily covered statements that have been repaired (Y axis) against the number of flaky tests that have been repaired (X axis). Lines that climb quicker (i.e., are drawn towards the top-left corner of the grid fastest) are 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 that Elbaum et al. [23] proposed for evaluating test case prioritization (i.e., the weighted average of the percentage of faults detected).

5.3 Evaluation Results

In all of the studied cases, GreedyFlake achieves the top prioritization performance. Figure 5.2 shows the Alberg diagrams where different approaches are compared.

Table 5.1 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



(a) Nova

(b) Neutron

(c) Cinder

Figure 5.2: 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 5.1: GreedyFlake Evaluation: AUCEC of various test case prioritization techniques

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.

5.4 Chapter Summary

In this chapter, we focus on the repair of flakily covered program elements. Particularly, we focus on the prioritization of flaky tests to repair. We present GreedyFlake, a greedy approach to prioritizing flaky tests for repair in order to fix flakily covered program elements. At every iteration, GreedyFlake re-ranks the set of flaky tests based on the number of flakily covered statements that can be robustly covered by repairing the flaky test. We evaluate GreedyFlake with various test case prioritization approaches proposed in literature.

Specifically, we compare GreedyFlake with an empirical random baseline, total statement coverage prioritization and additional statement coverage prioritization. We also evaluate if a re-ranking step is necessary at every iteration. We find that 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.

Chapter 6

Threats to Validity

Being based on empirical studies, this thesis is subject to threats to its validity. In this chapter, we describe those threats with respect to external, internal and construct validity types.

6.1 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 flakiness 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.

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.

6.2 Internal Validity

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

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.

6.3 External Validity

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

In our study, we rely on the data collected through the project's CI system to obtain the list of flaky tests. Many projects may not have a well-defined testing process or a mature CI

pipeline that captures test execution history. Efficient detection of flaky tests from source code is an evolving research area and various tools [7], [21] are being actively developed. Such tools can be adopted for projects without a well-defined testing process.

Chapter 7

Conclusion

In this chapter, we conclude this thesis by summarizing its contributions and presenting directions for future research.

7.1 Contributions and Findings

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. However, tests have varied characteristics that can be used to enrich coverage reports and help developers make informed decisions to improve the test suite quality.

First, we set out to enrich code coverage with test scope and flakiness. We perform an empirical study on three OpenStack projects and generate an enriched coverage reports. From the reports generated, we glean insights such as:

- More statements are covered only by unit tests than only by functional tests.
- 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.
- On the surface, two of the projects studied (Nova and Neutron) have greater coverage than Cinder. However, Cinder has the lowest proportion of flakily covered statements.
- Enriched code coverage measurements can effectively pinpoint which areas of the code base need improvement.

Next, we further analyze the flakily covered statements identified through the enriched coverage report and find that they cannot be explained through basic code characteristics, such as dispersion, ownership and developer activity.

Finally, since flakily covered statements are likely too numerous to address all at once, we shift our focus to the prioritization of flakily covered statements for repair. We propose GreedyFlake, a greedy approach to such prioritization. GreedyFlake ranks and selects the flaky test for repair that fixes the maximum number of flakily covered statements. At every iteration, GreedyFlaky re-ranks to select the test that fixes the maximum number of flakily covered statements. We evaluate the effectiveness of GreedyFlake by comparing with other baseline approaches. We find that 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.

7.2 Opportunities for future research

Below, we outline what we believe to be promising avenues for future research.

Visualize enriched coverage in code coverage tools

In this thesis, we explore enriching code coverage with test characteristics. We expand code coverage from a binary covered-or-not perspective, to multiple categories providing insights about tests covering the code. These categories can be included in the reports of code coverage tools and be highlighted in development views, such as code reviewing interfaces or even IDEs. Future work that evaluates the impact of visualizing the enriched code coverage on development and code base characteristics would likely be fruitful.

Explore other test characteristics that can be used to enrich coverage

In this thesis, we focus on two test characteristics, namely, scope and flakiness. Similar studies could be conducted based on other test characteristics. For example, Grano et al. [36], find that test cases tend to be more effective when they do not contain test smells [37]. Test

suites can be analyzed for test smells and the enriched coverage report can highlight when the program elements are covered by a test with smells.

Expand to other types of coverage

In this thesis, we focus on statement coverage. There are other types of coverage, such as branch coverage, condition coverage or function coverage. Enriching these types of coverage can be useful as well. For example, if a functional coverage report is enriched with test flakiness, it can be used to analyze if a particular function is prone to flaky testing.

Explore other code characteristics to explain incidences of flakily covered elements

While we determine that flakily covered program elements cannot be trivially determined through basic code characteristics, there can be other confounding factors that can explain the occurrence of flakily covered elements. Other code characteristics such as complexity and readability can be explored.

Repair prioritization to balance multiple objectives

In our evaluation of GreedyFlake, we find that a greedy approach can be beneficial while prioritizing flaky tests for repair. We assume that the effort required to repair all flaky tests are equal. However, flakiness is due to various reasons and can require varying levels

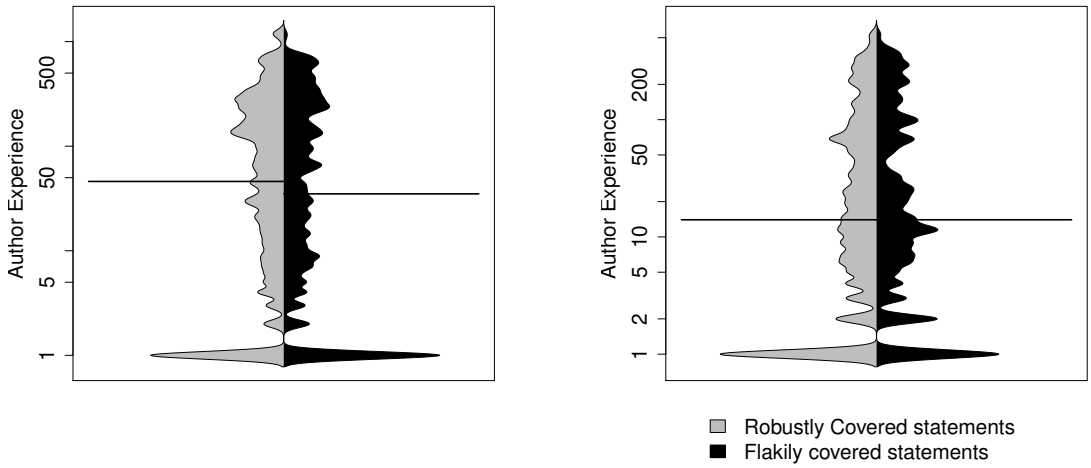
of effort to fix. If the effort required to produce a fix can be quantified, then prioritization approaches to repair flakily covered program elements can be re-evaluated as a multi-objective optimization problem.

Appendices

Appendix A

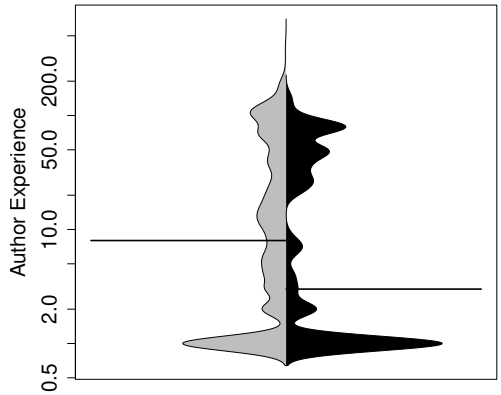
Additional Figures

This appendix includes bean plots that are used to visually compare various characteristics. The vertical curves of beanplots summarize and compare the two distributions. The higher the frequency of a particular experience value, the thicker the bean at that particular value on the y axis.



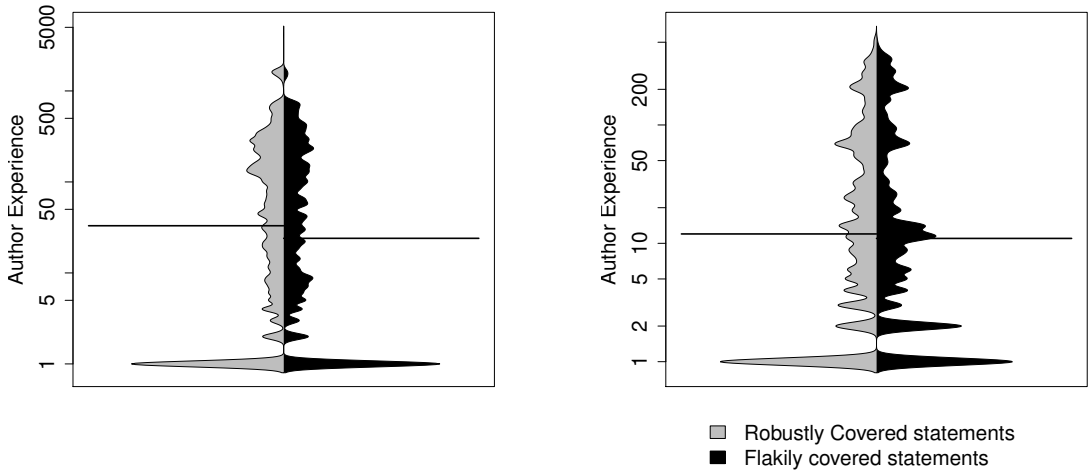
(a) Nova

(b) Neutron



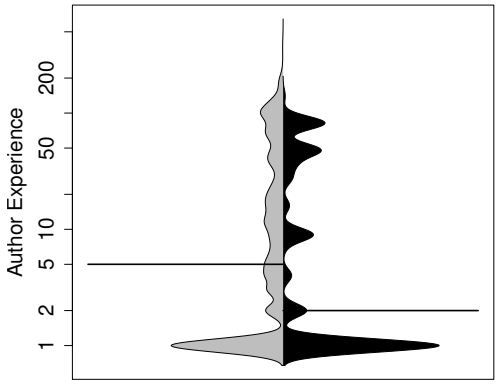
(c) Cinder

Figure A.1: Comparison of experience values of last-known authors of robustly covered vs flakily covered statements. We cannot distinguish between robustly covered and flakily covered statements based on the experience of the last-known author of the statement.



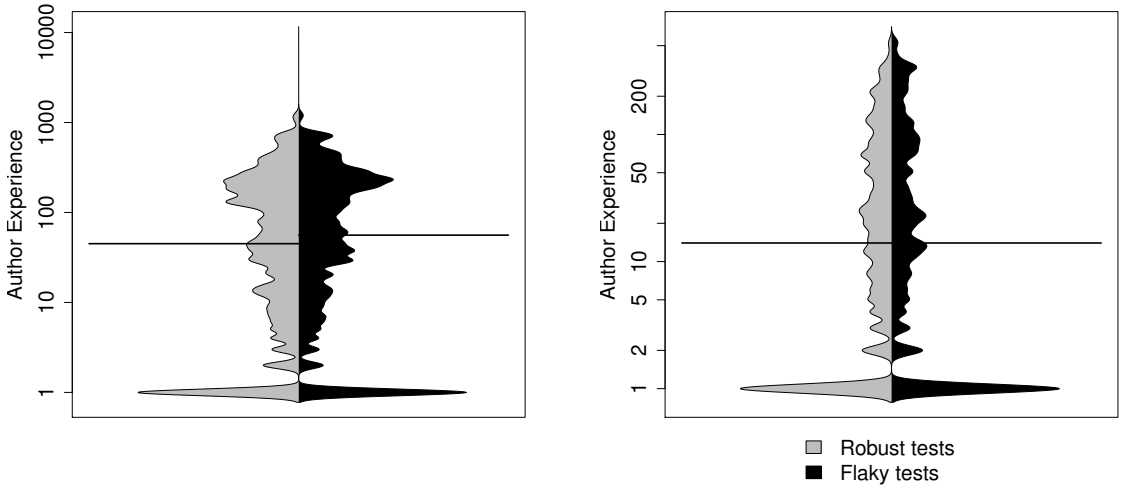
(a) Nova

(b) Neutron



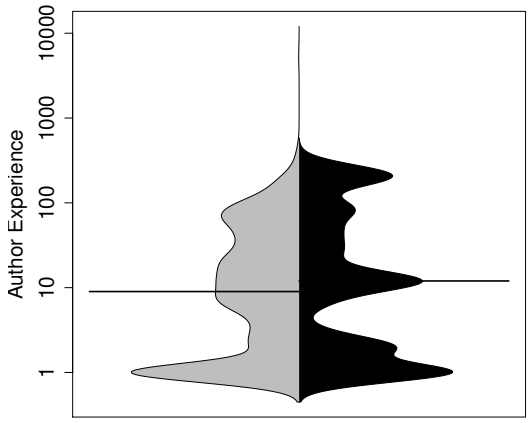
(c) Cinder

Figure A.2: Comparison of experience values of all authors for robustly covered vs flakily covered statements. We cannot distinguish between robustly covered and flakily covered statements based on experience values of all the authors of a statement.



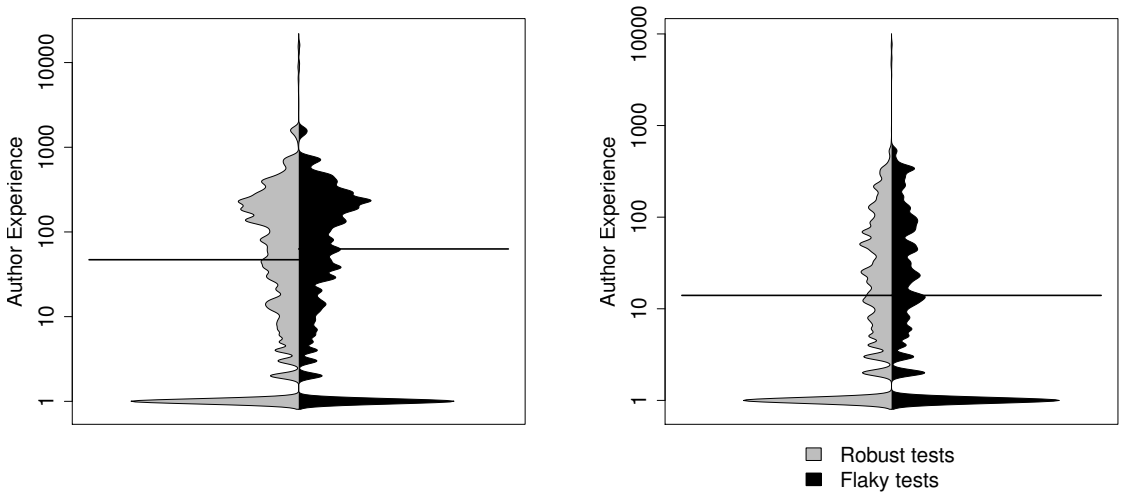
(a) Nova

(b) Neutron



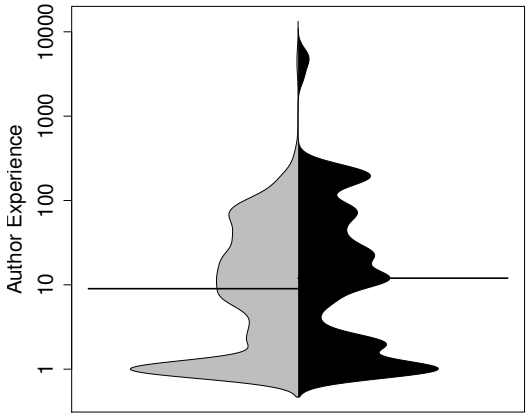
(c) Cinder

Figure A.3: Comparison of experience values of last-known authors in flaky vs non-flaky tests. We observe negligible differences in all three projects.



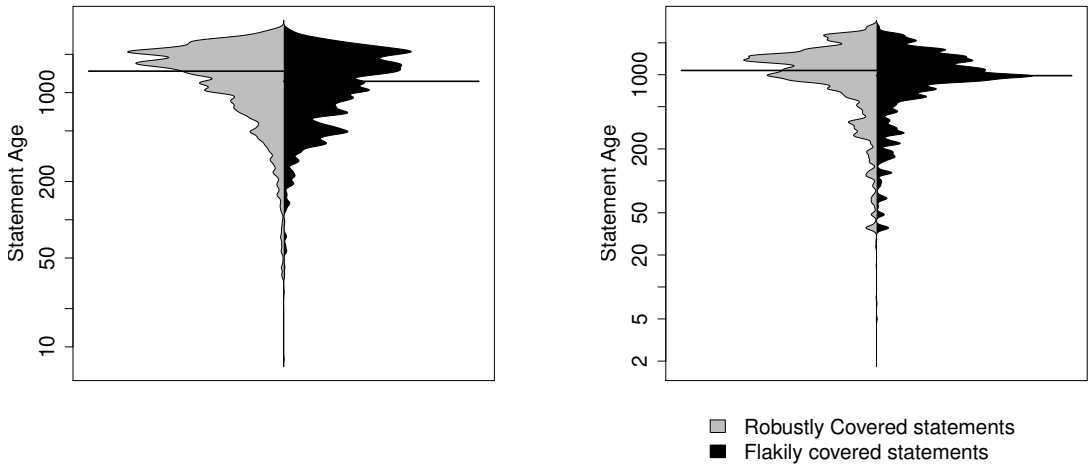
(a) Nova

(b) Neutron



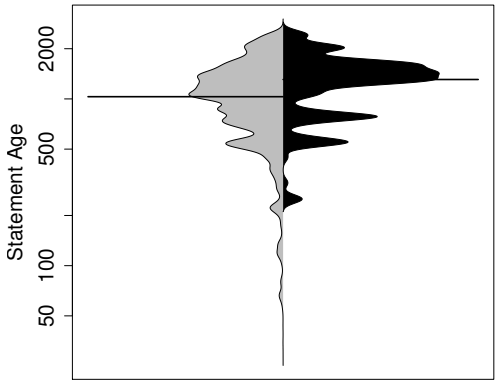
(c) Cinder

Figure A.4: Comparison of experience values of all authors in flaky vs non-flaky tests. We observe negligible differences in all three projects.



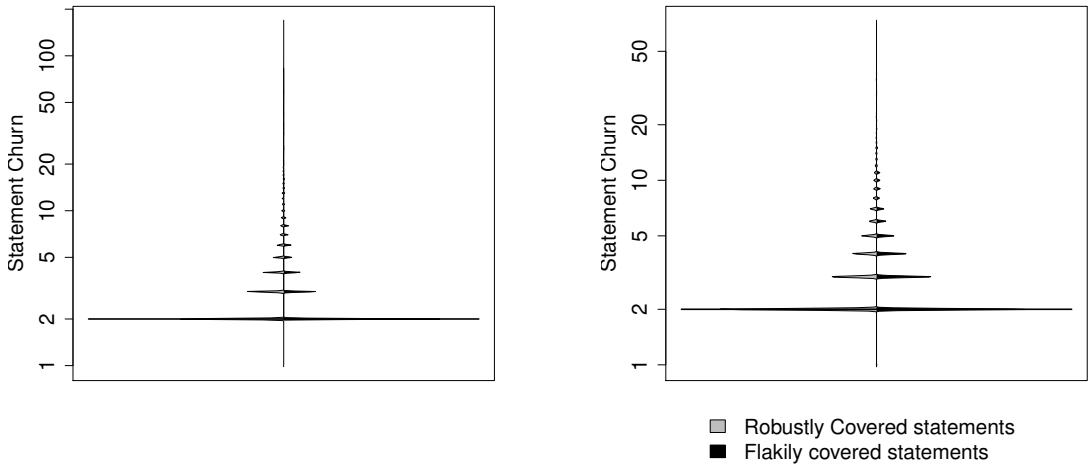
(a) Nova

(b) Neutron



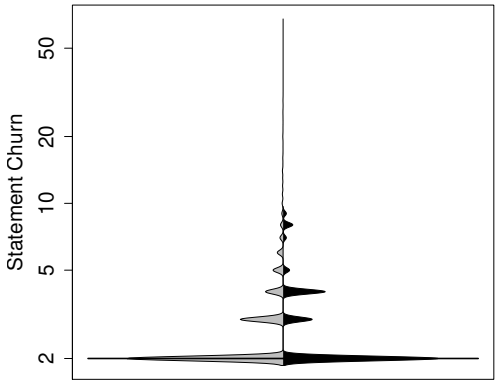
(c) Cinder

Figure A.5: Comparison of age of robustly covered vs flakily covered statements. Flakily covered statements are similar to robustly covered statements in terms of age.



(a) Nova

(b) Neutron



(c) Cinder

Figure A.6: Comparison of churn of robustly covered vs flakily covered statements. Flakily covered statements are similar to robustly covered statements in terms of churn.

Bibliography

[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.

[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*, pp. 435–445, ACM, 2014.

[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*, pp. 72–82, ACM, 2014.

[5] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[6] M. H. Halstead *et al.*, *Elements of software science*, vol. 7. Elsevier New York, 1977.

[7] 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*, pp. 643–653, ACM, 2014.

[8] 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.

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

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

[11] 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)*, p. 170, IEEE Computer Society, 2001.

[12] 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*, pp. 57–68, ACM, 2009.

[13] 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)*, pp. 345–355, IEEE, 2017.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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, pp. 449–456, IEEE, 2007.

[19] 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)*, pp. 534–538, IEEE, 2018.

[20] 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*, pp. 433–444, ACM, 2018.

[21] 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)*, pp. 312–322, IEEE, 2019.

[22] 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)*, pp. 100–107, IEEE, 2018.

[23] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 102–112, 2000.

[24] 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*, pp. 113–124, IEEE, 2004.

[25] 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.

[26] 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.

[27] 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.

[28] 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*, pp. 112–122, 2019.

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

[30] B. Shneiderman, "Tree visualization with tree-maps: 2-d space-filling approach," *ACM Transactions on graphics (TOG)*, vol. 11, no. 1, pp. 92–99, 1992.

[31] 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*, pp. 491–500, 2011.

[32] 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*, pp. 1–33, 2006.

[33] 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.

[34] 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.

[35] 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.

[36] G. Grano, F. Palomba, and H. C. Gall, "Lightweight assessment of test-case effectiveness using source-code-quality indicators," *IEEE Transactions on Software Engineering*, 2019.

[37] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*, pp. 92–95, 2001.