Investigating the Relationships Between Mutants, Oracles and Pseudo-Testedness

Megan Elizabeth Maton

Supervisor: Professor Phil McMinn

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

in the

Faculty of Engineering
School of Computer Science

March 25, 2026

Declaration

All sentences or passages quoted in this document from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure.

Megan Elizabeth Maton
March 2026

Abstract

Test suites are critical in developing high-quality software, yet are commonly measured using code coverage. Code coverage measures the quantity of test suite-executed code, without requiring code behaviours be checked by an oracle (e.g., an assertion), leaving code vulnerable to pseudo-testedness. Pseudo-tested code is executed by the test suite, yet can be deleted without triggering test failures. Intuitively, pseudo-testedness can be detected using deletion operators, but has not been explored at the statement level. The relationship between pseudo-tested statement identification (PTSI) and other oracle-based adequacy metrics, i.e., checked coverage, remains unclear, leaving open questions about each technique's role. Because PTSI relies on rule-based deletion mutation, it only highlights the absence of a check. To move beyond identifying where checks are missing, mutants must produce semantically incorrect behaviours that reveal the context requiring the check. While large language models (LLMs) enable context-aware mutation testing, current LLM-based tools are constrained to small code changes, such as lines or AST nodes. To utilise the LLM's capabilities, tools must move towards algorithm-level approaches that guide the LLM to produce non-trivial, productive mutants. This thesis aims to characterise these relationships and utilise them to refine the techniques. First, this thesis explores pseudo-testedness across 23 projects, identifying that pseudo-tested methods miss 48% of pseudo-tested statements. Secondly, compared with checked coverage using dynamic and observation-based slicers, PTSI locates code with the lowest median mutation scores (0.32 vs. 0.76 and 0.5) while executing at least 10x faster. Finally, this thesis uses systematic hazard analysis to guide LLMs in generating algorithm-level mutants that are uniquely subsuming and defect-matching compared with existing LLM- and rule-based techniques. By characterising the relationships between mutants, oracles and pseudo-testedness, this thesis enables the refinement of these techniques for test suite assessment.

Publications Resulting from this Research

The core contributions of this thesis are based on the following four papers, published at peer-reviewed venues:

M. Maton, G. M. Kapfhammer, and P. McMinn. Exploring Pseudo-Testedness: Empirically Evaluating Extreme Mutation Testing at the Statement Level. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2024

M. Maton, G. M. Kapfhammer, and P. McMinn. PseudoSweep: A pseudo-tested code identifier. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), Tool Track*, 2024

M. Maton, G. M. Kapfhammer, and P. McMinn. Where Tests Fall Short: Empirically Analyzing Oracle Gaps in Covered Code. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2025

M. Maton, G. M. Kapfhammer, and P. McMinn. Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2026

Each chapter that is based on a paper will state the author's contribution to the original paper and how the publication is used.

Acknowledgements

Firstly, I would like to thank my supervisor, Phil McMinn, for his continuous support and encouragement, and Gregory M. Kapfhammer for his ongoing support, guidance, and enthusiasm throughout my PhD. I would also like to thank the ASET research group for their support, laughter and entertainment over the last few years. In particular, I would like to thank Michael Foster for being a genuine friend and providing many delicious meals and de-stress walks to keep me going. Finally, I would like to thank my parents, Debbie and John, and my brother, Liam, for their love and many cups of coffee powering this thesis, as well as all my friends and family both in Sheffield and back home. Most of all, I thank my partner, Georgie, for being my rock and believing in me every step of the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to insufficient time allocation and limited recognition of testing effort [180], it is understandable that developers struggle to motivate themselves to test their code. Meeting metric targets, such as code coverage, gives developers a clear goal to work towards. But with limited time for testing, focusing solely on maximising these metrics' scores may result in other test weaknesses emerging. Therefore, developers need knowledge of the relationships among different approaches in order to make informed decisions. This thesis identifies these unexplored relationships (marked using **R**) and forms hypotheses and research questions to evaluate them.

1.1 Test Metrics (*and the problem with coverage*)

Code coverage (i.e., statement or branch coverage) is the most common metric for assessing test quality in industry, helping developers, e.g., at IBM, to design new test cases and improve existing ones [164]. When tests cannot reach certain areas of code, it also raises the question of code redundancy. More recently, developers at Google reported that code coverage helps to prompt testing when authoring code, but is even more valuable when reviewing code changes, as it enables them to quickly assess whether tests consider potential edge cases [88].

Structural code coverage is generally lightweight to compute, with many open-source tools — such as JaCoCo for Java — supporting variations including statement, branch and condition coverage [7, 14, 204, 209]. However, each of these techniques possesses known limitations [68]. Practitioners often use statement coverage as an indicator of test suite quality; yet, such criteria only require that the specified code elements are executed, leaving code vulnerable to be "pseudo-tested" — that is, code that a test suite executes, yet is entirely deletable without the test suite noticing [68, 109, 137]. For example, in Figure 1.1, the test suite covers Lines 12 to 17. If the method's functionality is completely removed and the test suite is rerun, it will continue to pass. It is therefore difficult to argue that this code has been well tested.

Pseudo-tested Method Example

Software under Test (SUT)

```
1  class ShoppingBasket {
2    private List<Double> items = new ArrayList<>();
3    private int rewardPoints = 0;
4
5    public void addItem(double price) {
6      items.add(price);
7      updateRewards(price);
8    }
9
10   // PSEUDO-TESTED METHOD
11   private void updateRewards(double price) {
12 -     if (price >= 100.0) {
13 -       rewardPoints += (int) (price * 0.10);
14 -     } else if (price >= 50.0) {
15 -       rewardPoints += 5;
16 -     } else {
17 -       rewardPoints += 1;
18 -     }
19   }
20
21   public int getItemCount() {
22     return items.size();
23   }
24
25   public int getRewardsCount(){
26     return rewardPoints;
27   }
28 }
```

Unit Test Suite

```
1  class ShoppingBasketTest {
2
3    @ParameterizedTest
4    @ValueSource(doubles = {120.00, 70.00, 30.00})
5    public void testAddItem(double price) {
6
7      ShoppingBasket basket = new ShoppingBasket();
8      basket.addItem(price);
9
10     // Covers all 3 branches of updateRewards,
11     // but does not check reward points.
12     assertEquals(1, basket.getItemCount());
13   }
14
15   // Test Suite only checks the item count is
16   // updated. There are no test cases
17   // targeting the rewardPoints.
18
19 }
```

Figure 1.1: Example of a pseudo-tested method using the `updateRewards` method and a parameterised test case that achieve 100% statement coverage of the method. +/- indicate added/removed lines of code for evaluating pseudo-testedness.

Existing techniques for finding pseudo-tested code are very coarse-grained, e.g., extreme mutation testing (XMT), which removes entire method bodies, as shown in Figure 1.1. While effective at the method-level, the impact of pseudo-testedness at the statement level is unknown. Naturally, it raises the question of how rule-based statement deletions, as used in mutation testing (see Section 1.3), can be combined with code coverage to identify pseudo-tested statements. Therefore, the first relationship this thesis investigates is:

R1 The relationship between pseudo-testedness and rule-based statement deletion.

This thesis hypothesises that XMT misses finer-grained pseudo-testedness and, therefore, statement deletion can be used to identify missed pseudo-testedness at the statement level. This, therefore, leads to research questions exploring their frequency, mutation scores, comparisons with traditional mutation testing and their causes. Identifying such statements requires a thorough test suite with strong assertions to confirm test outcomes, and therefore, pseudo-tested elements must guide developers towards this. While identifying pseudo-tested statements targets missing assertions, other techniques exist to measure oracle-based adequacy and therefore, their relationships with pseudo-tested statements should be considered.

Checked Coverage Example

Software under Test (SUT)

```
1 public int points (int win, int draw, int lose) {
2   // Executed, but not on the dynamic slice
3   totalGames = (win + draw + lose);
4
5   // Executed, and on the backward slice
6   int points = (win * 3) + draw;
7   return points;
8 }
```

Unit Test

```
1   @Test
2   public testPointsCalculation() {
3     int wins, draws, losses = 10, 5 , 2;
4     int points = points(wins, draws, losses);
5
6     // Assertion is the slicing criterion
7     assertEquals(35, points);
8   }
```

Figure 1.2: Example of a checked coverage evaluation using the `points` method and one test case, `testPointsCalculation`.

1.2 Oracle-based Adequacy Criteria

Oracle-based test adequacy techniques (e.g., checked coverage and state coverage) evaluate coverage from the perspective of the oracle, e.g., an assertion [113, 175, 193, 208].

Checked coverage uses a dynamic slicer to calculate which statements in the code-under-test directly influence the outcome of an assertion [175]. Where a statement is executed but does not appear on a dynamic slice from an assertion, it is labelled as uncovered, as it does not impact an assertion. Figure 1.2 presents an example of checked coverage using a sports team points calculator based on their wins, draws and losses. It also updates the external variable `totalGames` to reflect the team's total number of games played. However, despite the unit test `testPointsCalculation` achieving 100% code coverage, the `totalGames` variable is not a data or control dependency of the assertion (i.e., does not impact the variable values checked by the assertion) and therefore, is labelled as uncovered by checked coverage.

State coverage, on the other hand, evaluates the proportion of output defining variables that the test oracles evaluate, where output defining variables are all code-under-test (CUT) defined variables that are available to the test after CUT execution [113]. Both checked coverage and state coverage, however, come with significant barriers to practical use. Initial descriptions for both approaches require dynamic slicing, which is difficult to implement accurately, leaving few options with significant limitations. State coverage opted for dynamic taint analysis due to limited available options, resulting in state coverage executing 70x slower than the original test runner [112]. Checked coverage, on the other hand, opted for JavaSlicer, which had limitations, such as the inability to trace native Java libraries (e.g., `System.arraycopy()`) and the Java `String` library or to handle concurrent execution, resulting in incomplete slices [175]. Approaches such as these provide a metric for which statements are checked by assertions, not just for execution as part of a test, but are often hindered by implementation challenges.

Identifying *oracle gaps* — code that is executed by tests (i.e., covered) but does not cause a test case to pass or fail — can offer a cost-effective way to find where tests fall short [80]. Oracle gaps are calculated by combining code coverage with an oracle-based

Mutation Testing Example

Software under Test (SUT)

```
  public boolean canVote(int age) {
-     if (age >= 18) { // Original: Includes 18
+     if (age > 18) {  // Mutant: Excludes 18
          return true;
      }
      return false;
  }
```

Unit Tests

```
@Test
public void testAdultCanVote() {
    assertTrue(canVote(20)); // Mutant Survives
}

@Test
public void testBoundaryAge() {
    assertTrue(canVote(18)); // Mutant Killed
}
```

Figure 1.3: Rule-based Mutation example using the `canVote` method and two test cases, `testAdultCanVote` and `testBoundaryAge`. `+/-` indicate added/removed lines of code.

adequacy criterion to identify elements that are covered but not checked. Several oracle gap calculation approaches (OGCAs), such as host checked coverage, extreme mutation testing (XMT) and identifying pseudo-tested statements have emerged, seeking to highlight covered code that is unchecked by an oracle (e.g., a test assertion) with the goal of providing developers with clear testing direction without the expense of other techniques such as mutation testing [80, 129, 137]. However, the lack of empirical evidence for the role of pseudo-tested statements as an OGCA prevents developers from making informed choices on the most appropriate way to evaluate their test suite. The second relationship this thesis investigates is, therefore:

R2 The relationship between pseudo-testedness and oracle-based adequacy criteria.

Due to the inefficiencies and inaccuracies of checked coverage implementations, this thesis hypothesises that pseudo-tested statement identification produces a more useful oracle gap than checked coverage. This raises research questions exploring the prominence, distribution, statement categorisations, fault-detection abilities and performance of each technique's oracle gaps. By using rule-based statement deletion mutation, pseudo-tested statement identification highlights *where* the assertion checks miss, but does not consider the *context* in which they are needed. However, the introduction of Large Language Models (LLMs) into mutation testing enables context-aware mutation and no longer relies solely on rule-based mutation operators.

1.3 Context-aware Mutation Testing

Mutation testing measures fault-detection capability by injecting synthetic faults (i.e., mutants) into program code to determine whether the test suite detects them, thus helping developers identify missing test inputs and oracles [95]. Traditional rule-based mutation testing is based on the competent programmer hypothesis [48]. That is, programmers do not write code at random. Through iterative design and improvement, especially with modern review processes, program code is often *close* to being correct, with most faults

stemming from missing control paths, inappropriate path selection and inappropriate or missing actions [48, 66]. As such, traditional rule-based mutation uses a set of predefined code transformation patterns to apply changes to the code-under-test, as a way of mimicking such faults [95]. For example, a relational operator replacement transformation would "mutate" the code `a == b` to `a >= b`, resulting in a potential control flow change in the code-under-test (CUT). If a test case were to fail as a result of this change, the mutant is detected and therefore said to be "killed". However, if no test case fails, the mutant "survives" and highlights an area that may require further testing effort. For example, in Figure 1.3, the operator `>=` is mutated to `>`, which excludes 18-year-olds from voting. The first test case `testAdultCanVote` checks the value `20` and therefore does not fail on the mutant code. The second test case `testBoundaryAge` checks the value `18` and fails on the mutant version, therefore "killing" the mutant. If a test suite can detect fabricated faults, there is strong empirical evidence that it is more likely to detect real faults [15, 98].

Although widely acknowledged as an effective measure of real fault-detection ability, traditional rule-based mutation testing remains prohibitively impractical [15, 16, 95]. This is primarily due to the considerable computational expense (e.g., mutant generation and execution) and the manual effort required to triage equivalent mutants [44, 56, 98, 165]. To address these challenges, researchers have explored various cost-reduction strategies to improve the practical viability of mutation testing [95, 142, 165]. By understanding the characteristics of a "productive" mutant, researchers can avoid the generation of unproductive ones and move towards only producing mutants that offer value to a developer [163]. A mutant is considered *productive* if it is either killable by the test suite, or functionally equivalent to the original code yet can provide insights that will improve knowledge or code quality [163]. Therefore, to avoid developers experiencing information overload due to a low ratio of productive mutants, it is critical that research reduces the mutant volume and improves mutant quality to ensure efficient and effective mutation testing. While many existing rule-based operators are coupled with specific fault types, there are fault types, such as algorithm modification, numerical analysis errors and similar method-call replacement, that cannot be captured with simple rule-based transformation patterns [98].

The recent introduction and advancements of large language models (LLMs) and their tuning for coding tasks have enabled context-aware code mutations that bridge this gap [187]. However, existing techniques use LLMs to produce small changes to individual lines or the AST, similar to rule-based approaches, and therefore fail to replicate algorithm-level faults [36, 187]. As such, to generate algorithm-level mutants, we must explore controlled methods to replicate developer misunderstandings via semantic-based mutations. This thesis proposes that hazard analysis can guide these semantic transformations and constrain the LLM to generate productive method-level mutants. Therefore, the final relationship this thesis investigates is:

R3 The relationship between hazard-guided LLM-based mutants and LLM-/rule-based mutants.

This thesis hypothesises that using hazard analysis processes from safety engineering can guide an LLM-based mutation process to create defect-matching and subsuming mutants. This raises research questions exploring mutant properties, subsumption relationships, defect-matching between techniques and whether the results hold across both local and cloud-based LLMs. By guiding the LLM to introduce method-level mutations through hazard analysis, these mutants can rewrite the method algorithm to represent a developer misunderstanding rather than a syntactic "typo".

By addressing these research questions to test each hypothesis, this thesis bridges the gaps between mutants, oracle gaps and pseudo-testedness, characterising the relationships between the techniques to expose missed test suite weaknesses.

1.4 Aims

The primary aims of this thesis are to explore and characterise the relationships between mutants, oracles and pseudo-testedness, and to use this characterisation to refine techniques that improve their utility. To achieve these aims, this thesis is guided by the following high-level objectives:

1. To **characterise** the relationships between mutants, oracles and pseudo-testedness, specifically through identifying missing assertions or test cases and triggering fault-revealing test cases.

2. To **refine** techniques for producing productive mutants, identifying oracle gaps and evaluating pseudo-testedness.

1.5 Thesis Contributions

To achieve these aims, this thesis makes the following contributions: each chapter explores and empirically evaluates one of the previously stated relationships. Figure 1.4 presents the links between each chapter.

1.5.1 Pseudo-testedness at the Statement Level (Chapter 3)

Chapter 3 forms part of the paper I have published in the Proceedings of the 40th International Conference on Software Maintenance and Evolution (ICSME), 2024 [129].

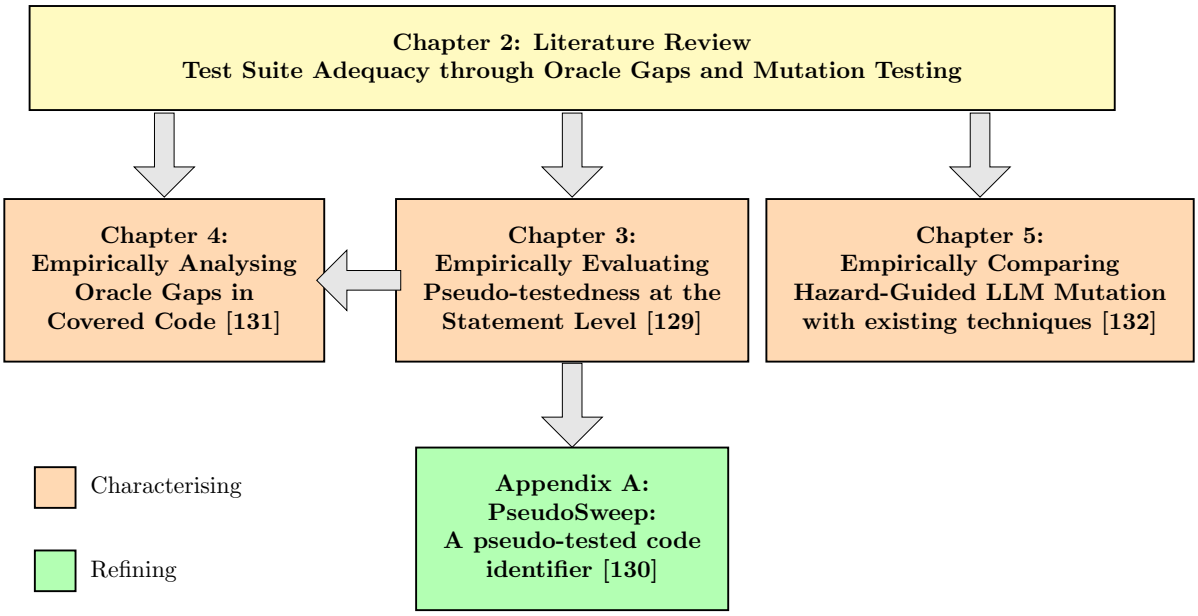

Figure 1.4: The links between the chapters in this thesis: orange blocks focus on "characterising" the relationships between mutants, oracles and pseudo-testedness, and green blocks target "refining" techniques to produce them. Citations indicate where a chapter is based on published work.

As discussed in Section 1.1, research on XMT to date has focused on pseudo-testedness in Java Methods, leaving it unexplored at the statement level. To characterise pseudo-tested statements, Chapter 3 uses and extends the statement deletion (SDL) operator to identify statements that are pseudo-tested by a test suite, finding 722 examples of pseudo-tested statements across four frequently-studied, large, Apache Commons Java projects, as well as 23 projects randomly selected from the Maven Central Repository [2], the largest official software repository for Java (**R1**). Chapter 3 presents an empirical study, finding that: (i) XMT misses 48% of the pseudo-tested statements as they appear in methods it classifies as required; (ii) pseudo-tested statements achieve lower mutation scores than required statements and (iii) a manual classification revealed most pseudo-tested statements are due to specific test suite inadequacies (e.g., missing tests or partial assertions) that XMT and traditional mutation testing cannot highlight due to their inapplicable operators. The contributions of this chapter are therefore as follows:

A1. A quantitative evaluation of the frequency of pseudo-testedness at the statement level, specifically those left undetected by XMT (Section 3.5.2).

A2. A quantification of the differences between pseudo-tested statements and other covered statements using traditional mutation scores (Sections 3.5.3 and 3.5.4).

A3. A qualitative manual analysis of the causes of pseudo-tested statements to understand the test suite inadequacies that they uncover (Section 3.5.5).

1.5.2 Empirical Comparison of techniques for identifying Oracle Gaps (Chapter 4)

Chapter 4 is based on a paper I have published in the Proceedings of the 19th International Symposium on Empirical Software Engineering and Measurement (ESEM), 2025 [131].

This chapter builds on the findings of Chapter 3 to characterise the role in finding executed but unchecked code that pseudo-tested statements play, when considering their relationships with other oracle gap techniques, namely checked coverage using a dynamic slicer and checked coverage using an observational slicer (**R2**). To enable comparison of the three techniques, this chapter first establishes the "generalised" oracle gap definitions. It then quantifies the gaps calculated by each approach across 30 Java classes in six open-source projects to understand both their sizes and the extent of their overlap. A low overlap between techniques suggests they may be complementary, offering different information to the developer; however, if the content is similar, it is valuable to examine the advantages of one approach over the other. Subsequently, the chapter presents a manual examination of the statement types that appear in each gap and their relationship with a test suite's fault-detection ability. Finally, Chapter 4 examines the execution time of each approach to assess its scalability and practical applicability in developer workflows.

This study found that, in general, developers cannot use the three techniques interchangeably. In terms of detection capability, PTSI highlighted code locations exhibiting the lowest median mutation scores (0.32), compared to CC_{DS} (0.76) and CC_{OS} (0.50). This indicates that PTSI is particularly effective at revealing high-priority areas where fault detection is weakest. A manual inspection identified data loading, iteration and output updates as the predominant statement patterns within the oracle gaps across the approaches. In terms of efficiency, PTSI required the shortest median execution time of 19.9 seconds—significantly faster than 273.2 seconds (CC_{DS}) and 5957.1 seconds (CC_{OS}). Consequently, this balance of high-priority targets and fast execution suggests that PTSI is a good initial test suite analysis approach for strengthening weak areas. This chapter makes the following contributions:

B1. A generalised definition of the oracle gap. Existing techniques use disparate terms, such as "coverage gap" and "pseudo-tested statements". This chapter presents a unified definition to enable comparative analysis of techniques. (Section 4.2)

B2. A quantitative evaluation of the oracle gap prominence, distribution, fault detection rate and execution times using checked coverage with dynamic and observational slicing and pseudo-tested statement identification (Sections 4.4.1–2 and 4.4.4–5).

B3. A qualitative manual inspection of code patterns identified as ineffectual by each technique (Section 4.4.3).

1.5.3 Method-level mutation guided by Hazard Analysis via LLMs (Chapter 5)

Chapter 5 is based on a paper that has been accepted for publication in the Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE) 2026 [132].

This chapter introduces hazard analysis techniques as a mechanism for guiding LLMs to produce method-level mutants and empirically evaluates their relationship with rule-based and LLM-based mutation approaches (**R3**). To fully utilise the LLM's code-generation potential and semantic reasoning capabilities, this approach extends LLM-mutation to the method level; however, as with any LLM-based generation task, providing sufficient guidance is essential to provoke a contextually relevant response. This study introduces a three-stage technique in which the LLM derives descriptions of the method under test, then applies the HAZOP (Hazard and Operability Study) and STPA (Systems-Theoretic Process Analysis) hazard analysis techniques to these descriptions to identify possible deviations (HAZOP) or unsafe control actions (STPA) [108, 119]. The LLM then implements each deviation or unsafe control action as a mutant of the method under test.

To evaluate the role of LLM-generated hazard analysis mutants, the chapter presents an empirical comparison with three LLM-based mutation techniques and one rule-based technique to characterise their role in mutation testing. The study answers research questions that evaluate: the mutant properties (e.g., compilability); subsumption relationships between techniques; defect-finding test case triggering; and a manual survey of the mutants to understand the kinds of mutations the hazard analysis techniques lead to. This study found that the unguided method mutation produced the most syntactically divergent mutants that the test suites killed trivially, necessitating a guided approach, such as hazard analysis, to produce method-size mutants. This was also reflected in the subsumption relationships and defect analysis. In terms of subsumption relationships and unique defect triggering, mutant set size appeared to influence outcomes: the larger the mutant set of an individual technique, the more likely it was to have uniquely subsuming or defect-finding mutants. When sampled, each LLM-based technique performed more similarly, with both the HAZOP and STPA techniques contributing a significant proportion of the uniquely subsuming and defect-finding mutants. Upon downsampling to the smallest set size, the performance gap between the LLM-based techniques narrowed, revealing that both HAZOP and STPA generate substantial numbers of uniquely subsuming and defect-finding mutants even when only a subset of mutants is used. As such, the contributions of this chapter are as follows:

C1. A Hazard-guided mutation technique, leveraging LLMs to generate code descriptions, apply hazard analysis and mutate Java methods to implement risky be-

haviours (Section 5.2).

C2. An empirical study to understand the role of hazard-guided mutation relative to state-of-the-art rule-based and LLM-based techniques (Section 5.3).

1.5.4 PseudoSweep: A pseudo-tested code identifier (Appendix A)

Appendix A forms part of a tool paper I have published in the Proceedings of the 40th International Conference on Software Maintenance and Evolution: Tool Demo Track (ICSME — Tool Demo Track), 2024 [130].

To enable the study of pseudo-tested statements in Java methods, Appendix A presents the open-source tool, PseudoSweep, which uses and extends the statement deletion operator to identify pseudo-tested code (**R1**). This tool instruments the Java code using a metamutant to ensure all mutants are compilable. The pseudo-tested methods and statements identified in Chapters 3 and 4 were found using PseudoSweep. Appendix A contains further implementation details regarding the tool and a case study on the *language.Caverphone2.encode* method from *commons-codec* discussed in Section 3.5.5. The contribution of this chapter is therefore:

D1. The PseudoSweep tool for using code-deletion mutants and statement coverage to identify pseudo-tested methods and statements (Section A.2).

1.5.5 Summary of Contributions

In summary, this thesis fulfils the research aims set out in Section 1.4 by characterising the relationships between mutants, oracles and pseudo-testedness, and by extending this to refine and empirically evaluate the contributory techniques. To characterise **R1**, Chapter 3 empirically evaluates how rule-based deletion operators can be adapted (**D1**) and used (**A1**–**3**) with code coverage to expose pseudo-tested statements beyond pseudo-tested methods as implemented in PseudoSweep in Appendix A. Chapter 4 then contributes **B1**–**3** along with **D1** to characterise **R2**, specifically the role pseudo-tested statements play in revealing oracle gaps and the differences and similarities with existing techniques. Finally, Chapter 5 contributes **C1**–**2** to refine LLM-based mutation techniques using hazard analysis and characterise their relationships (**R3**) with existing LLM- and rule-based techniques. Collectively, these contributions advance knowledge of mutation testing, oracle gaps and pseudo-testedness in test suite assessment and refine techniques to improve their value to developers.

Chapter 2

Literature Review

Use of Publications in this Chapter Sections of this chapter are based on the publications supporting subsequent chapters to ensure this literature review provides a complete background with definitions to support this thesis. To maintain a consistent voice throughout this thesis, I have independently re-authored any sections drafted by publication co-authors. Furthermore, sections of this literature review have previously been submitted as part of the Confirmation Review process for this award.

2.1 Introduction

Software testing is fundamental to the development of high-quality software [14, 127, 135]. This chapter establishes the theoretical foundations and terminology essential for the research presented in this thesis. It begins by discussing the role of software testing and core terminology, then defines the code-based metrics used in this thesis. Specifically, these include code coverage criteria (e.g., statement coverage); oracle-based adequacy criteria (e.g., checked coverage); and, finally, the definition of relevant mutation testing terminology, including pseudo-tested code. Section 2.5 further maps out the field of mutation testing, including tools, techniques and industrial practices, which add context for the contributions of this thesis.

2.2 Software Testing

Software testing is a crucial part of software development, primarily used to evaluate the quality of the software under test [14]. In practice, this involves executing a software under test (SUT) using test cases composed of sets of inputs targeting specific path executions and evalutions of their outcomes against expected results, which collectively form a test suite [14, 127, 209]. However, a fundamental limitation of software testing is that it can only show the presence of faults, and cannot guarantee their absence [39]. The main goal

Software Under Test (SUT)

```
1  public String getExtension(String filename) {
2    // FAULT: Developer mistakenly uses
3    // indexOf instead of lastIndexOf
4    int dotIndex = filename.indexOf('.');
5
6    if (dotIndex == -1) {
7        return "";
8    }
9    return filename.substring(dotIndex + 1);
10 }
```

Unit Test

```
1  @Test
2  public void testExtractsFinalExtension() {
3    // Input forces Infection & Propagation
4    String result = getExtension("backup.tar.gz");
5    String expected = "gz";
6
7    // Revealability: expected vs actual
8    assertEquals(expected, result,
9       "Should extract the final extension");
10 }
```

Figure 2.1: An example SUT method with respective unit test including a fault that is detected by the unit test.

is therefore to create a test suite of SUT inputs that finds as many faults as possible. The consequences of an insufficient test input set can range from minor issues, such as user annoyance, to major concerns, such as data loss or theft, service outages, or, in safety-critical systems, even loss of life. To achieve this, developers must test software at multiple levels to find and address faults as early as possible [14, 20].

2.2.1 Unit Testing

Unit testing is the "testing of individual hardware or software units", targeting specific behaviours in the SUT [6]. When the program's output is not as expected, this usually indicates a fault (i.e., a static defect) in the SUT [14]. The most cost-effective time to find and address faults is during the initial design or unit testing phases. Consequently, agile paradigms such as *Extreme Programming (XP)* prioritise practices such as test-driven development (TDD) to ensure defects are identified as early in the lifecycle as possible [18, 19]. While testing the system as a whole is important, testing at a finer granularity has two main benefits: first, providing confidence in the individual components before combining them; and second, if a fault is detected by a unit test, the fault location must occur within a specific module, therefore localising debugging efforts [135]. A test case, therefore, must be capable of detecting faults.

The example in Figure 2.1 shows a method in the SUT that identifies the final file extension of a filename. The unit test checks the "ExtractsFinalExtension" behaviour to ensure that where multiple *dots* appear in the file name, only the final one is used to extract the extension. The unit test specifies an expected outcome and uses the `assertEquals` method to verify that the actual result matches it. The test currently fails because it successfully exposes a fault in the SUT. The mechanics of a fault exposed through a failure are formalised using the Fault Detection Model [14].

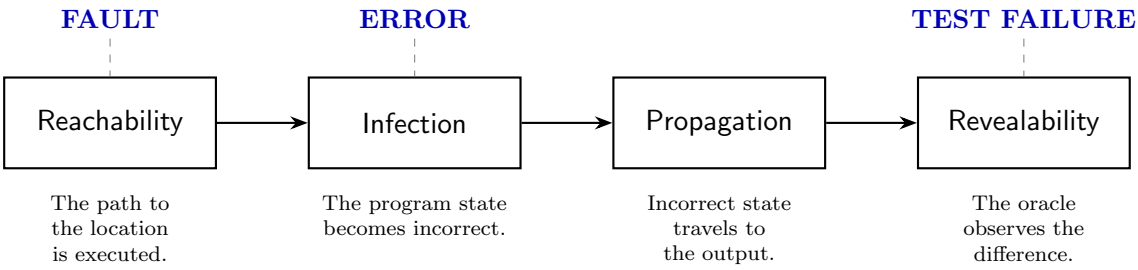

Figure 2.2: The RIPR Model Criteria (adapted from Ammann and Offutt [14]).

2.2.2 Fault Detection Model

The RIPR (Reachability, Infection, Propagation, Revealability) model identifies the conditions under which a fault can be detected [14]. As outlined in Figure 2.2, the four steps require that the test case must:

1. **Reach the fault**: The test inputs execute the path to the fault.
2. **Infect the state**: The test inputs cause the fault to create an incorrect state.
3. **Propagate the incorrect state**: The incorrect state continues to the output.
4. **Reveal the incorrect state**: A test oracle observes the incorrect state.

When using code coverage alone, a tester can only confirm that a fault is reachable with no guarantee that it will infect the state and propagate to the output. We can only reveal incorrect outputs and observe failures by adding checks to verify program output. Such checks are known as test oracles and often take the form of an assertion statement (e.g., Lines 8–9 in Figure 2.2) in software testing [17]. In the example in Figure 2.2, the fault on Line 4 of the SUT calls the `indexOf` method on `filename`, which returns the index of the first instance of the character passed in via the parameters. However, in this example, to identify the extension, the code should return the substring after the final ".". Using the test input string "`backup.tar.gz`", the fault is reached, and it infects the state by resulting in an incorrect value (i.e., an error) for the variable `dotIndex`. The incorrect value of `dotIndex` then propagates through to the output by causing Line 9 in the SUT to return the incorrect substring. Finally, the fault is revealed by the unit test assertion on Lines 8–9, which compares the expected result against the actual result and fails because they are not equal.

There are many test adequacy criteria and testing principles that developers can use to help guide them towards a *good* test suite [27, 192]. However, each criterion satisfies different components of the fault detection model and, as such, cannot provide confidence that the test suite is "effective". In this thesis, the term "effective test suite" refers to a test suite that can detect faults in the software under test. All results from experiments executed in Chapters 3, 4 and 5 concern the unit tests of the software under test.

2.3 Code Coverage

Code coverage is a widely used metric that helps developers identify which parts of a codebase are executed by a given test suite [14]. These structural testing metrics count the execution of elements such as statements, branches and conditions [76, 209].

2.3.1 Control Flow-based Code Coverage

In 1963, Miller and Maloney set out the objective of finding a complete set of "possible", not "potential", test cases for a given SUT through various input conditions proposed as *systematic mistake analysis* [134]. Although then state-of-the-art, Miller and Maloney's work reflects the substantially smaller and less complex nature of systems at the time. Yet, this work set the foundation for today's branch coverage. Statement coverage, however, is the most commonly used metric among both researchers and developers due to its simplicity, representativeness and ease of understanding. Yet, there is conflicting evidence regarding its effectiveness [68, 76, 86, 208]. It is usually represented by the percentage statements executed by a test suite, that is:

$$\text{Statement Coverage Score} = \frac{|\text{Executed Statements}|}{|\text{Total Statements}|} * 100 \tag{2.1}$$

For safety-critical systems, developers employ stronger coverage criteria, such as object-branch coverage (OBC) and modified condition/decision coverage (MC/DC), to ensure systematic test suite development [33, 38]. OBC requires that each conditional jump instruction is both taken and not taken in the compiled binary. While similar to source-code branch coverage in theory, OBC generally requires stronger test cases to execute each instruction. MC/DC, on the other hand, requires that each source code conditon and decision be executed true and false, and that each condition individually flips the overall decision. However, due to their strict requirements, these are rarely adopted outside these domains.

Other techniques for applying further conditions to coverage have evolved, such as direct coverage, which considers methods covered only if they are directly called from a test case [84]. Unfortunately, such additional techniques are often released only as research prototypes rather than as complete tools, thereby limiting practical adoption. Practical tooling is easy to use and responsive, with Java tools such as JaCoCo providing a visual representation of coverage [7], and SimpleCov for Ruby taking this further by providing the "hit/line" rate, showing how many tests execute a given line [145].

Companies such as Google use JaCoCo, among other tools, within their developer workflows to consistently measure unit test code coverage [88]. Ivanković et al. surveyed Google's code coverage integration, recommending that in order to integrate code coverage into industrial workflows, coverage should be measured automatically by existing estab-

lished libraries and integrated into tools the development team are currently using [88]. Ivanković et al. further extended this to create "Productive" coverage at Google, which prioritised specific element coverage if similar code was also well covered [89]. However, this depends on the project's historical code coverage, which requires substantial internal setup. As a result, it creates a significant leap from research prototypes to industrial adoption.

2.3.2 Data flow-based Code Coverage

Data flow code coverage criteria evaluate how variable definitions and uses affect program execution [58]. Rapps et al. derived a set of coverage criteria from data-flow analysis techniques, including all-defs, all-uses and all-def-uses, with the latter being one of the strongest [166]. All-defs-uses requires every path between every variable definition and use to be executed by the test suite. All-uses, on the other hand, requires only one path between each definition and use to be executed [209]. While data flow-based code coverage techniques are generally considered to be stronger than the control flow-based techniques discussed in Section 2.3.1, their computation is more expensive [172].

However, relying solely on code coverage guarantees only that a test case reaches the fault location. It does not ensure that the specific test inputs will cause the fault to infect the program state, propagate to the output, or ultimately be revealed. Therefore, stronger adequacy criteria are necessary.

2.4 Oracle-based Adequacy Criteria

Test oracles (e.g., the JUnit assertion on Line 8–9 in Figure 2.1) determine whether a test case passes or fails and are vital in assessing expected program behaviour [14, 17, 179]. However, distinguishing correct and incorrect program behaviour is a challenge known as the oracle problem [17]. Test oracle choice is an important factor in the effectivess of a test suite, and it is therefore important to find techniques to help developers identify the right assertions for their tests [64]. Since the number of assertions demonstrates a strong positive correlation with test suite effectiveness (i.e., fault finding), identifying missing oracles is core to improving test suites [208].

2.4.1 State Coverage

Distinct from traditional state-based testing, state coverage in this context determines whether unit tests check program outputs and side effects [113]. As such, a test suite is *state coverage inadequate* if a program statement specifies outputs that do not impact test assertion outcomes [112]. State coverage was one of the first coverage-based techniques to enforce output checking in unit tests, measuring the extent to which the test suite checks

the program behaviour. State coverage uses a control flow graph (CFG) in which each node represents a statement and is labelled with all variable definitions and references within that statement. The CFG is determined by program slicing to identify statements and their relationships to the test oracles [200].

Program Slicing

Program slicing locates the statements with potential consequences (all data and control dependencies) on variables used at a specified location, known as a slicing criterion [73, 199, 200]. A *static slice* includes all statements with potential consequences — including those that may not be of immediate interest — whereas *dynamic slicing* constrains this to a given set of inputs [111]. State coverage employs static slicing, using variables checked by the test oracles as the slicing criterion. An output-defining variable/statement is covered if it appears on at least one slice [79].

A core problem with state coverage has been its implementation. Initial implementations using a static slicer found state coverage to provide insights that did not require further human analysis, as with mutation testing (see Section 2.5), but were less sensitive to the number of checks than mutation testing. This could likely have been improved using dynamic slicing; however, implementation challenges limited a future version to dynamic taint analysis [35, 136]. A taint uses a program's control and data flow to identify where values of interest are used by other values, with taint analysis placing markers as it goes. The assertion in a test acts as a "sink": the evaluation point for the "tainted" values. Koster's implementation of state coverage proved to be around 70 times slower than the standard JUnit test runner. Another extension on state coverage provided a generalised algorithm but did not provide enough feedback to help developers debug their code [193]. Due to the challenges of implementing state coverage, it is not practical beyond research examples.

2.4.2 Observable Coverage

To force control flow coverage techniques to be more effective at finding faults, multiple techniques apply an observability criterion to existing coverage metrics to enforce the *propagation* and *revealability* criteria of the Fault Detection Model as shown in Figure 2.2.

For instance, while MC/DC is considered one of the strongest coverage criteria, it suffers from the same issues as other code coverage metrics: it does not fulfil the criteria for fault detection described in Section 2.2.2. Observable MC/DC (OMC/DC) coverage ensures that the effects of faults would be revealed by oracles [201]. By adding a path condition to MC/DC, OMC/DC requires stronger oracles in existing test cases rather than increasing the number of test cases. As a result, requiring the effects to propagate through the rest of the SUT execution to an observable output variable resulted in up to

88% higher fault detection [201]. However, with even more intensive requirements than MC/DC, OMC/DC is not practical for non-safety-critical domains.

More generally, Observable Structural Coverage has been extended to other structural criteria such as branch coverage, condition coverage and decision coverage [133]. Each of these observability metrics uses a forward tagging system in which the tag must pass from the criterion to the test oracles without being masked (i.e., another condition prevents the outcome of the first condition from being identified at the point of the test oracles).

Each of these techniques proved stronger at fault detection than their coverage-only counterparts; however, they led to longer tests and bloated test suites, often unsuccessfully, in an attempt to fulfil the test criteria. Issues such as complexity and dead code made achieving observable coverage difficult.

2.4.3 Checked Coverage

Instead of extending a control-flow metric, checked coverage uses a dynamic backwards slice from an assertion to identify the statements that cause an oracle to pass or fail [174, 175].

Dynamic Slicing

As described in Section 2.4.1, static program slicing locates the statements with potential consequences (all data and control dependencies) on variables used at a specified location. A backward slice is computed by selecting a slicing criterion and traversing the control and data dependencies backward, identifying all statements that may influence the outcome of the criterion. The criteria, in this case, are the test oracles (e.g., JUnit assertion statements like Lines 8–9 of the Unit Test in Figure 2.1). Dynamic slicing constrains the slice to locate only the statements that actually impact the variables given specific inputs [111]. The reduced number of statements means dynamic slices (each representing a single execution) are usually significantly smaller than static slices. Using this definition, checked coverage is therefore defined as follows:

Definition 2.4.1 (Checked Coverage using Dynamic Slicing)**,** Checked Coverage using Dynamic Slicing requires each sliceable statement to be on at least one dynamic backward slice from an explicit test suite check.

To measure checked coverage, the checked coverage score is calculated as the percentage of executed statements in a dynamic backward slice.

$$\text{Checked Coverage Score} = \frac{|\text{Statements on at least one dynamic backward slice}|}{|\text{Executed Statements}|} \quad (2.2)$$

Initial implementations of checked coverage for Java used a dynamic slicer called JAVASLICER [9]. JAVASLICER first traces a program through bytecode instrumentation, creating a trace file, which it then uses to compute the slice. In practice, the tool calculates the checked coverage score as follows [174]:

1. Identify all check statements (e.g. JUnit assertions) and all coverable statements within the SUT.
2. Trace each test class separately to compute dynamic backward slices.
3. Merge the slices from the test classes to create a set of all statements that influence the checked values.
4. Calculate the checked coverage score using the number of statements in the set created in Step 3, divided by the number of coverable statements calculated in Step 1.

Empirical results have demonstrated that checked coverage is more sensitive to assertion removal and quality than mutation testing and statement coverage and is strongly correlated with test suite effectiveness [175, 208]. Checked coverage has also been used in Test Suite Reduction (TSR) using SLICER4J to instrument jar files and perform execution tracing and dynamic slicing [110]. However, compared to line and method coverage, checked coverage provides lower fault-detection capabilities in reduced test suites and significantly higher run times, making it ineffective for this task.

The original definition of checked coverage was not tied to a control flow coverage metric like the observable coverage metrics in Section 2.4.2. Hossain et al. generalised the checked coverage definition to apply to a *host metric* and defined the *coverage gap* as the percentage difference between host metric coverage (h) and host metric checked coverage (hcc) [80]. They found that the greater the coverage gap, the lower the fault detection rate, and used this to identify "focus methods" that lacked assertions. They further developed a static analysis technique based on host checked coverage output and focus methods to provide testers with testing recommendations. While this development has improved checked coverage and expanded its application, it still incurs substantial overhead. This overhead is significant for developers using such systems in their workflows and is something future work must address to enable practical application.

Checked coverage relies on computing a dynamic backward slice. However, dynamic slicing is challenging to implement, as evidenced by the scarcity of tools — for Java, the only notable options are SLICER4J and JAVASLICER, each with critical limitations [11, 9]. JAVASLICER, for example, can only view the bytecode of the classes loaded by the JVM. Therefore, the tracer cannot see code called via the Java Native Interface, which many methods in the Java Standard Library use for inputs and outputs. As a result, the tracer cannot observe parts of the program execution and therefore loses dependencies, leading

to a lower-than-expected checked coverage score. The tracer also cannot handle the String class (amongst other integral Java classes, including System and Object). Being limited in this way reduces the usefulness of checked coverage, as it cannot cover a significant proportion of Java code. Decoupling this definition and studying other slicing techniques in checked coverage may yield differing results.

Observational Slicing

In Chapter 4, this thesis studies the use of observational slicing — an observation-based slicing derivative — to calculate checked coverage. *Observation-based slicing* is a language-independent technique that constructs a program slice by collectively deleting lines and observing program behaviour [24, 65]. If the line can be deleted and the original behaviour of the slicing criterion (i.e., the state of a variable at a given location) remains, the deletion is accepted. If the deletion changes program behaviour under the given slicing criterion, the deletion is rejected.

Rather than analysing data and control dependencies like other slicing techniques [186], the observation-based slicing implementation, known as ORBS, *deletes* lines, *executes* the remaining candidate slice and *observes* the behaviour for the given slicing criterion. By deleting lines rather than specific statements, ORBS achieves language independence, as it requires no syntactic or semantic knowledge of any programming language. ORBS can therefore also slice programs written in multiple programming languages, unlike dynamic slicing techniques. Although observation-based slicing and dynamic slicing are related, there is a key difference: Observation-based slices are *observed* dependencies, whereas dynamic slices contain statically obtained but dynamically triggered dependencies [25]. This can lead to dependencies that are statically determinable but may not be observable. Therefore, dynamic slicing is an inherent under-approximation because potential dependencies can be observed but do not correspond to any statically determined dependence.

Observational slicing generalises observation-based slicing to a variety of observable behaviours beyond variable values [65]. This generalisation enabled the slicing technique to apply to tree-structure representations of models. This enables test suite/test case outcomes to serve as slicing criteria for observable behaviour, i.e, passing/failing [65]. Fundamentally, this generalisation also enables it to use slicing criteria that dynamic slicers cannot, as these require criteria to be specified at locations within the source code [9]. For instance, dynamic slicing checks the variable's state at each assertion, whereas observational slicing can use the overall test suite outcome as its criterion.

2.4.4 Oracle Identification and Improvement

While the techniques discussed above highlight where oracles are missing or weak, there is ongoing research into how to generate and improve test oracles.

Oracle Data Selection

Identifying the optimal set of program variables to observe during software testing helps to reduce cost of testing while maintaining test quality [121]. The DODONA approach, for example, automates this process by observing the interactions and dependencies between variables to propose a set of variables that engineers should monitor using test oracles [121]. Oracle data selection can then be combined with mutation testing to rank variables in terms of fault-finding effectiveness to further define the most valuable test oracles [64]. By looking at the variables that are most sensitive to faults, the tester can then identify a small number of expected values to create a concise set of effective oracles.

Oracle Quality

To enable successful oracle generation, it is important to be able to define and assess oracle quality. Therefore, understanding characteristics that can make an oracle brittle, and being able to identify them can help to improve long term oracle quality. A brittle assertion can be described as one that checks variable values that are derived from inputs that the test does not control [83]. As such, these tests can become difficult to maintain and fail when changes are made to code that is not responsible for the behaviour under test. On the other hand, it is also therefore important to ensure that the behaviours that are derived from the inputs controlled by the test are checked by assertions. The ORACLEPOLISH approach identifies both brittle assertions and unused inputs at a "reasonable cost" by using input tainting and input mutation to assess which inputs are checked and not checked by the assertions.

Oracle completeness and oracle soundness are important factors of oracle quality. Oracle completeness requires all correct program states to be accepted, whereas oracle soundness requires all faulty states to be rejected [90]. It is, therefore, critical to identify false positives (i.e., accepted faulty states) and false negatives (i.e., rejected correct states) to increase developer confidence in generated oracle quality. An empirical study found that human developers exhibit poor performance when manually performing this task [92]. The ORACLE ASESSMENT AND IMPROVEMENT (OASIs) tool uses search-based testing techniques to find counter examples that demonstrate incompleteness and unsoundness, and provide developers with a decision support tool to help improve oracles [90, 91]. Such approaches still rely on developers to make the final decision on expected behaviour, as this is not something a search-based tool can do due to seeing only implemented behaviours.

Despite being designed to improve fault detection, oracle-based test adequacy techniques cannot independently assess their effectiveness in achieving this goal. Therefore, developers use mutation testing to simulate fault detection [15, 48].

2.5 Mutation Testing

Despite its name, mutation testing is not a software testing technique but a method for evaluating the fault-detection strength of test suites. Mutation testing introduces syntactic changes, called mutants (e.g., a flipped operator), into a program's source code to evaluate a test suite's ability to find real faults if they were present [15, 48, 95]. Mutation testing is motivated by the principle that *all programmers are competent* [48], and therefore defects introduced into code by programmers are likely to be minor mistakes or "typos". The *coupling effect* states that test inputs that can distinguish subtle faults in the SUT should also be able to distinguish the more complex errors as well, and has been supported by multiple studies [48, 138, 139]. By inserting such mistakes into the code, we can evaluate a test suite's ability to identify them. As a result of the coupling effect, we can address more complex issues by first targeting minor defects. Therefore, mutation testing fulfils the Fault Detection Model (see Section 2.2.2) RIPR criteria by inserting a fault that must cause an error and be revealed by a test oracle.

2.5.1 Definitions

To evaluate test suite effectiveness, mutation testing uses the following terminology to classify the synthetic faults and their outcomes.

Definition 2.5.1 (Mutant)**,** a simple syntactical change OR a program containing a simple syntactical change.

Definition 2.5.2 (Killed Mutant)**,** a mutant that yields a test outcome that differs from an un-mutated code's test outcome.

Definition 2.5.3 (Surviving Mutant)**,** a mutant that yields the same test outcomes as an un-mutated code.

Definition 2.5.4 (Equivalent Mutant)**,** a surviving mutant that does not semantically change the code; therefore, a test case cannot detect the mutant.

Mutants that cause the test suite to fail are said to be *killed*, as the test suite has successfully identified them. However, mutants undetected by the test suite have *survived* and show a shortcoming in the test suite. As a result, the surviving mutants show where testing needs improvement or where a mutant is *equivalent* to the original program.

Mutation Operators

Mutation operators are the types of transformations a tool may apply to the SUT to create mutants.

Definition 2.5.5 (Mutation Operator)**,** a transformation rule for applying mutants.

```
-  return filename.substring(dotIndex + 1);
+  return filename.substring(dotIndex - 1);
```

(a) AOR: Arithmetic Operator Replacement (Killed)

```
-  return "";
+
```

(b) SDL: Statement Deletion (Surviving)

```
-  if (dotIndex == -1) {
+  if (dotIndex != -1) {
```

(c) ROR: Relational Operator Replacement (Killed)

```
-  if (dotIndex == -1) {
+  if (dotIndex == 0) {
```

(d) LVR: Literal Value Replacement (Surviving)

Figure 2.3: The (a) AOR, (b) ROR, (c) SDL and (d) LVR mutation operators applied to correct version (i.e., using `lastIndexOf`) of the `getExtension` method in Figure 2.1. Added/removed lines are prefixed with `+` / `-`.

Traditional rule-based operators typically insert, delete or replace code elements in the SUT. Examples of rule-based mutation operators used in tools such as MAJOR and PIT include:

- **Arithmetic Operator Replacement (AOR)**: Replace binary arithmetic operators such as "`+`" with compatible alternatives such as "`-`" (see Figure 2.3a).
- **Statement Deletion (SDL)**: Removes (or deletes) an individual statement or statement block (see Figure 2.3b).
- **Relational Operator Replacement (ROR)**: Replaces a relational operator such as "`==`" with compatible alternatives such as "`!=`" (see Figure 2.3c).
- **Literal Value Replacement (LVR)**: Replaces a literal value such as "`1`" with a compatible alternative such as "`0`" (see Figure 2.3d).

In the examples in Figure 2.3, we can see that two of the mutants were killed and two survived. This is due to the test case exploring only one instance of an extension and not considering edge cases, such as no extension or files starting with "`.`" (e.g., .gitignore).

There are many operators, each highlighting different types of deficiencies in the test suite. In particular, Chapters 3 and 4 focus on the utility of deletion-based mutation operators and are discussed further in Section 2.5.3. Depending on the language, tool and operator sets, some operators can produce mutants that may not be compilable. For example, Figure 2.3b depicts a full deletion of a return statement, which is compilable as an intermediate return, but not as the last return of the method.

Equivalent and Duplicate Mutants

Equivalent and duplicate mutants add to the cost of mutation testing, requiring additional human effort [205]. Equivalent mutants pose the biggest problem for mutation testing

```
-  if (dotIndex == -1) {
+  if (dotIndex <= -1) {
```

Figure 2.4: An equivalent mutant applied to the `getExtension` method in Figure 2.1. Added/removed lines are prefixed with `+` / `-`.

because detecting them is undecidable, as proven by Budd et al. [29]. An equivalent mutant may change the SUT syntactically; however, the outcome of the change cannot be distinguished from the original code. For example, the `indexOf` method used in the `getExtension` in Figure 2.1, returns either the index value of the "." or `-1` if the "." is not present. Therefore, applying the mutant in Figure 2.4 does not change the behaviour of the method, as the value of `dotIndex` cannot go below `-1`. Therefore, it is not possible to write a unit test case with inputs to differentiate between the mutant and the original behaviour.

These mutants are expensive, as they require human evaluation to determine whether a developer can write a test case to distinguish their behaviours. The equivalent mutant rate (EMR) [198] is calculated as follows:

$$\text{EMR} = \frac{|\text{Equivalent Mutants}|}{|\text{All Mutants}|} \tag{2.3}$$

A low EMR means fewer redundant mutants, potentially saving developer time. However, empirical results indicate that 10–40% of mutants may be equivalent, and automating their detection and reducing their generation is critical to improving the viability of mutation testing in practice [95, 140, 141]. Many techniques target equivalent mutants, attempting to avoid their generation or to automatically identify them [77, 106, 107, 118, 150].

Duplicate Mutants are "useless" mutants that exhibit the same behaviour as other mutants, making them equivalent to other mutants rather than the original SUT and many techniques applied to reduce and identify equivalent mutants have been applied to this problem [45, 56, 107, 126, 150].

Mutation Score

The mutation score measures the percentage of generated (and non-equivalent) mutants killed by a given test suite for the SUT [95, 165]. It is calculated by:

$$\text{Mutation Score} = \frac{|\text{Killed Mutants}|}{|\text{Generated Mutants}| - |\text{Equivalent Mutants}|} \tag{2.4}$$

Researchers use mutation score to compare test suites for a given SUT; however, this is primarily a research activity [207]. This is useful as a comparison measurement for test generation and reduction methods due to mutation analysis being recognised as an indicator for a test suite's fault detection ability. In practice, the meaning of the mutation

score is less clear. The individual mutants themselves are more meaningful to developers, particularly in terms of testability and observability of individual mutants. Pseudo-test suites (i.e., test suites either derived or generated for research) can further impact the mutation score [207, 210]. Furthermore, as shown in Equation 2.4, the mutation score should not consider equivalent mutants. However, due to the detection of equivalent mutants being undecidable, eliminating and accounting for all equivalent mutants is not feasible. Therefore, mutation scores often do not account for equivalent mutants and instead use only |Generated Mutants| as the denominator for the calculation [151].

2.5.2 The Relationship between Mutants and Faults

Mutation testing is assumed to simulate whether a test suite can detect real faults — formally known as the coupling effect. However, the relationship between mutants and real faults is an ongoing research topic with unclear outcomes [104, 153]. Early empirical studies by Andrews et al. found that mutation testing is a reliable indicator of fault detection [15, 16]. In contrast, Just et al. found that although existing operators coupled with real faults, many real faults do not couple with a mutant, requiring further work on fault-representative mutants [98]. To address this gap, multiple studies have examined real-world fault detection and compared testing outcomes with those of mutation operators. For example, Brown et al. analysed developer fixes in GitHub repositories to derive novel mutation operators, such as argument swapping, argument omission and a similar library method call that better reflects natural developer errors [28]. Most recently, Gay et al. investigated the degree of coupling between mutants and faults in the Defects4J dataset, comparing each defect's *trigger tests* (i.e., the tests that detect the real defect) and mutant-killing tests to identify the strongest mutation operators [63].

Defining the relationship between mutants and real-world faults remains an ongoing yet important challenge for researchers seeking to improve mutation testing operators.

2.5.3 Cost Reduction

Mutation testing is often seen as prohibitively expensive for practical implementation and use, and therefore many studies aim to reduce this cost [165]. For the last 25 years, studies have categorised these techniques into the following primary objectives: *do fewer*, *do smarter* and *do faster*, which relate to how mutation testing is performed [142]. However, the increased complexity of cost-reduction techniques has called for a new set of six primary objective categories, as shown in Figure 2.5 and explained below.

PG-1 Mutant Volume Reduction: The goal is to reduce the volume of mutants produced and executed without reducing effectiveness.

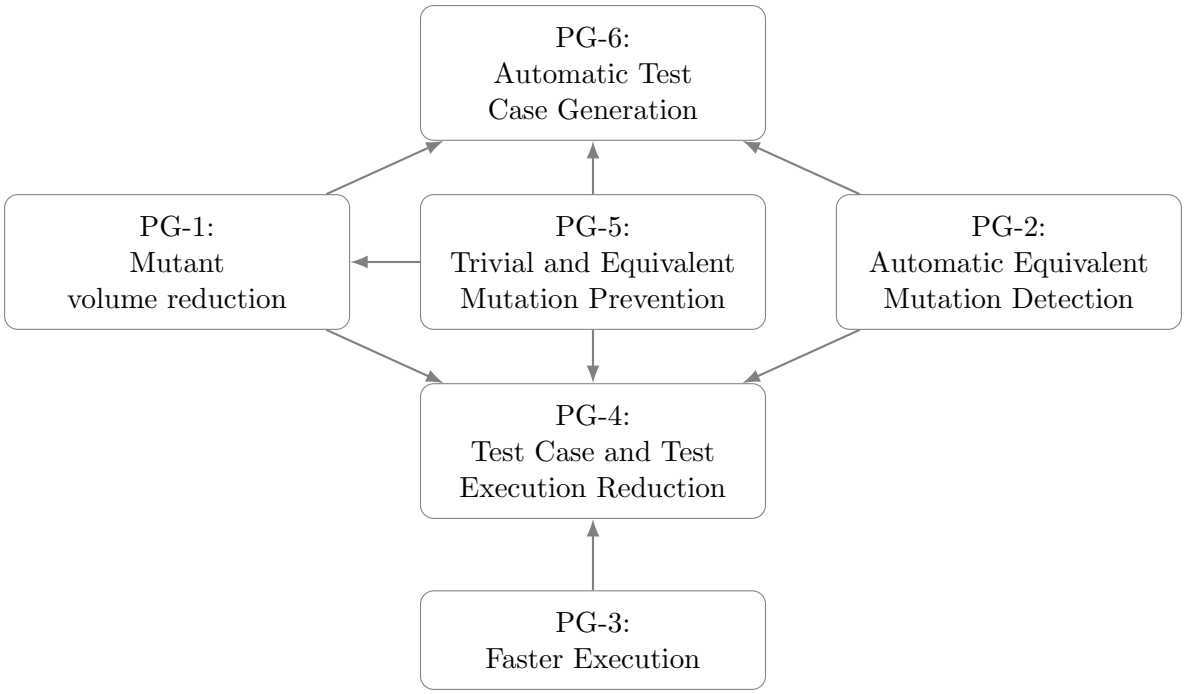

Figure 2.5: Cost reduction primary goals and the relationships between them. Figure adapted from Pizzoleto et al. [165]. Unidirectional arrows imply by achieving for example PG-3, you indirectly contribute to or achieve PG-4.

PG-2 Automatic Equivalent Mutant Detection: The goal of automating equivalent mutant detection and removing them from consideration before human evaluation.

PG-3 Faster Execution: The goal of improving algorithms, hardware and techniques for mutant execution to reduce mutant execution time.

PG-4 Test Case and Test Execution Reduction: The goal is to identify smaller groups of test cases that are effective at killing mutants, thereby reducing the number of test runs and execution time.

PG-5 Trivial and Equivalent Mutation Prevention: The goal of preventing the generation or execution of equivalent mutants by defining mutation operators or algorithms to generate fewer mutants.

PG-6 Automatic Test Case Generation: The goal of automatically generating test cases that kill as many mutants as possible, reducing the effort required for hand-writing test cases.

As cost reduction is a broad topic, the remainder of this section will discuss techniques for achieving PG-1, as they are most relevant to this thesis.

Reducing the number of mutants (PG-1) is the most extensively studied area of cost reduction [165]. Decreasing the volume of mutants and maintaining effectiveness reduces

```
-  a = b * c;
+  // a = b * c;
```

Figure 2.6: A single statement commented out using Deng et al.'s definition of the SDL operator for Java. Added/removed lines are prefixed with `+` / `-`.

Table 2.1: Default SDL return statements for specified return types (adapted from Deng et al. [49]).

Return Type	**Mutant**
Boolean	`return false` / `return true`
Double	`return 0`
Float	`return 0`
Char	`return 0`
Short	`return 0`
Int	`return 0`
String	`return null`
Long	`return 0`

both computational and manual analysis costs associated with mutation testing [165]. Researchers have applied many techniques to the problem, including (but not limited to): *random mutation* [67], *generation optimisation* [30], *data-flow analysis* [158], *control-flow analysis* [159], *selective mutation* [143], *minimal mutation* [13] and *sufficient operators* [177]. This literature review focuses on the *one-op mutation*, *higher order mutation* and *arid node removal* techniques, as Chapters 3 and 4 include *one-op mutation* and *arid node removal*, while Chapter 5 includes a manual analysis of mutants with a comparison to *higher order mutation* testing.

One-op Mutation

One-op mutation applies a single mutation operator to reduce the overall number of generated mutants while maintaining effective program coverage [190]. Primarily, studies have used deletion-based operators that remove code structures from the SUT to approach this challenge [44, 165].

The Statement Deletion Operator (SDL) The statement deletion (SDL) operator removes entire statement blocks from the program under test, as shown in Figure 2.3b and executes the test suite to see if it causes a failure [49]. The statement deletion mutant is "killed" if the test suite detects the deletion (i.e., a test fails) and it "survives" if it does not.

Deng et al. define how the SDL operator can be applied to Java programs, including deletion mutations for single statements (see Figure 2.6), `if`, `while`, `for`, `try-catch` and `return` statements. Table 2.1 shows the different SDL mutants that can be applied given

Required Method Example

Software under Test (SUT)

```
1  class ShoppingBasket {
2    private List<Double> items = new ArrayList<>();
3    private int rewardPoints = 0;
4
5    public void addItem(double price) {
6      items.add(price);
7      updateRewards(price);
8    }
9
10   // REQUIRED METHOD
11   private void updateRewards(double price) {
12 -     if (price >= 100.0) {
13 -       rewardPoints += (int) (price * 0.10);
14 -     } else if (price >= 50.0) {
15 -       rewardPoints += 5;
16 -     } else {
17 -       rewardPoints += 1;
18 -     }
19   }
20
21   public int getItemCount() {
22     return items.size();
23   }
24
25   public int getRewardsCount(){
26     return rewardPoints;
27   }
28 }
```

Unit Test Suite

```
1  class ShoppingBasketTest {
2
3    @ParameterizedTest
4  + @CsvSource({
5  +   "120.00, 12", // 120 * 0.10
6  +   "70.00,   5", // Flat reward
7  +   "30.00,   1"  // Base reward
8  + })
9    public void testAddItem(double price,
10                           int expectedPoints) {
11
12     ShoppingBasket basket = new ShoppingBasket();
13     basket.addItem(price);
14
15     assertEquals(1, basket.getItemCount());
16
17 +   assertEquals(expectedPoints,
18                    basket.getRewardsCount());
19   }
20
21 }
```

Figure 2.7: This example addresses the pseudo-tested method identifed using XMT in Figure 1.1 using the `updateRewards` method. The test case has been updated to check the `rewardPoints` and therefore making the `updateRewards` method "required" to make the test suite pass. This is because removing the method body will now cause the assertion on Line 16 to fail. +/- indicate added/removed lines of code for evaluating pseudo-testedness.

a `return` statement. Appendix A gives greater detail on how SDL mutants can be applied to the Java programming language. The SDL operator forms the basis of studying pseudo-tested statements in Chapter 3. Prior empirical studies have shown that, on its own, SDL produces significantly fewer mutants than traditional mutation testing, without sacrificing effectiveness for both Java and C programs [49, 50, 190]. The SDL operator typically uses the transformations described in Table 2.1 Delamaro et al. found that SDL is combinable with other deletion operators, such as operator and variable deletion, to enhance SDL's detection of testing concerns, before also extending the study from Java to C [44, 43]. However, Durelli et al. found that equivalent mutants generated by a statement-deletion operator are as easy to identify as those produced by traditional mutation operators [54]. The SDL operator is limited in its implementation across different Java mutation testing tools, as discussed in further detail in Section 2.5.5.

Extreme Mutation Testing Extreme mutation testing (XMT) reduces the number of mutants generated and executed by only removing method bodies from the SUT [137]. Where a return value is necessary for the program to compile, XMT adds default return values using preset values for different types. After each deletion, XMT runs the tests to determine whether the deletion causes a failure. These definitions classify elements

Partially-tested Method Example

Software under Test (SUT)

```
1  class ShoppingBasket {
2    private double total = 0.0;
3
4    public void addItem(double price) {
5      total += price;
6    }
7
8    // PARTIALLY-TESTED METHOD
9    public boolean isEligibleForDiscount() {
10     return total >= 100.0;
11   }
12
13 }
```

Unit Test Suite

```
1  class ShoppingBasketTest {
2
3    @Test
4    public void testEligibleForDiscount() {
5      ShoppingBasket basket = new ShoppingBasket();
6      basket.addItem(150.00); // total = 150.0
7
8      boolean eligible = basket
9                          .isEligibleForDiscount();
10     assertTrue(eligible);
11   }
12
13 }
```

Figure 2.8: Example of a **partially-tested** method. The test suite evaluates `isEligibleForDiscount()` with a total of 150. Mutating the return expression to constant `false` is **killed** (test expects `true`), but mutating it to constant `true` **survives**. Because the expression can be fixed to `true` without failing tests, it is partially-tested.

evaluated under XMT:

Definition 2.5.6 (Not-covered (N))**,** An element in the program under test that is not executed by the test suite.

Definition 2.5.7 (Required (R))**,** A program element that is executed by the passing test suite and that **cannot** be removed without impacting the test outcomes (see Figure 2.7).

Definition 2.5.8 (Pseudo-tested (P))**,** A program element that is executed by the test suite and that **can** be removed without impacting test outcomes (see Figure 1.1).

Definition 2.5.9 (Partially-tested (A))**,** A program element executed by the test suite that **can** be fixed to only a subset of possible values without impacting test outcomes (see Figure 2.8).

To illustrate the distinction between the last two definitions, consider a boolean variable, which may be mutated to "true" and "false". Killing only one mutant leaves the method (or statement) "partially-tested" [23], whereas if both mutants survive, the variable would be "pseudo-tested". The concept of pseudo-testedness is extended in Chapter 3 to study pseudo-tested statements.

Extreme mutation testing reduces the overhead of mutation testing by applying only a few operators (depending on the method return type) to each method. Previous studies used traditional mutation testing [48] to confirm that pseudo-tested methods are less thoroughly tested than other covered methods [23, 195]. These studies calculated method-level mutation scores for required and pseudo-tested methods, finding that the pseudo-tested methods had lower scores than those required by the tests. There are two existing tools for XMT in Java: Descartes and Reneri, which are discussed further in Section 2.5.5.

```
LOGGER.info("Extracting extension for file: " + filename);
```

Figure 2.9: An example of an **arid node**.

Higher Order Mutation

In Chapter 5, a manual analysis compares the LLM-generated mutants with higher-order mutants (HOMS). HOMS are compositions of first-order mutants (FOMS) — single-element transformations — that do not necessarily focus on specific program components, with the core idea being that a subsuming HOM is harder to kill than a single first-order mutant [69, 93, 94]. For a mutant m_1 to subsume mutant m_2, the tests that kill m_1 must be a subset of those that kill m_2. Therefore, killing m_1 is (a) harder than killing m_2, and (b) will also kill m_2 [114]. Strongly subsuming HOMS— HOMS whose killing test cases also kill all the individual FOMS as well — reduced the number of mutants by 35–45% and improved test effectiveness by 5.6–12% [74]. It is generally considered that HOMS are a cost-effective alternative to FOM rather than a replacement, as while significantly reducing the number of mutants, HOMS are less effective at revealing faults [152]. Without techniques for efficiency improvement, the search for meaningful HOMS is often computationally expensive [202].

Arid Nodes

One approach to reducing the number of mutants is to remove "arid nodes" from consideration. A program-under-test's abstract syntax tree consists of nodes that connect in a parent-child fashion to form a tree structure. Different nodes are of varying importance to a developer for testing. Petrović and Ivanković defined "arid nodes" as AST nodes that relate only to logging and testing, control only non-functional properties or are fundamental to the language, and higher order testing would trivially kill any mutants within them. Some statements, such as logging statements shown in Figure 2.9, are not crucial for developers to test. Mutation testing can avoid mutating these "arid" nodes as they are not a valuable use of resources [159] as referred to in Section 3.4.3.

2.5.4 Large Language Models (LLMs) for Mutation Testing

Natural Language Processing (NLP) and Programming Language Processing (PLP) have been significantly advanced by Large Language Models (LLMs) [194]. LLMs can perform tasks such as text generation, translation, code completion and repair, as well as code generation [34, 42, 120]. This rise in large language models for code-related tasks has naturally raised questions about their use in mutation testing. As with most current mutation testing research, many techniques aim to reduce costs, as shown in Figure 2.5; however, many studies conduct exploratory investigations because the field is still in its infancy.

Mutant Generation

Applying learning and generative approaches to mutant generation has enabled the generation of context-aware, or more "realistic", mutants.

Model training from errors and defects The recent rise in LLMs for mutation testing stems from a group of learning-based techniques that utilise deep learning models to learn fault patterns from historical bug fixes and common coding errors [157, 189]. Such techniques aim to introduce mutants that resemble real defects and issues, to improve the realism of mutation testing. For example, Tufano et al. trained models to generate mutants using learned bug fixes using the GitHub Archive in their tool DeepMutation [188, 189]. DeepMutation translates the original method into a "buggy" method token by token, whereas LEAM builds on this by using the method context to pinpoint where the method should be mutated before predicting the necessary grammar rules required and only the statement being mutated [184]. This improved fault syntactic correctness and better represented real faults. This is superseded by LEAM++, which uses mutant selection to improve mutant generation efficiency whilst maintaining the performance of LEAM. Such techniques proved effective, but required training purpose-built models; however, the pre-trained language models enabled faster implementation of context-aware mutation testing approaches.

Pre-trained Language Models Recent approaches using pre-trained language models (e.g., CodeBERT) and LLMs (e.g., GPT-4.1) have enabled mutation testing tools to benefit from context-aware, semantic mutation without the need to train their own models.

Firstly, the mask-and-predict style approaches hide a section of the code from the language model and ask it to suggest alternatives for the missing code. The μBERT uses CodeBERT to generate mutants using this approach. Test suites guided by μBERT mutants detect 67.5% of faults as well as improving cost effectiveness over PIT in terms of requiring fewer mutants to guide the test suite to identify more faults [41]. μBERT mutants have also been empirically shown to be able to generate mutants that are strongly coupled to real vulnerabilities, but this only accounted for 1.17% of the killable mutants [62]. As explained further in Section 2.5.5, LLMorpheus also uses the mask and predict approach, asking the LLM to suggest three alternatives for each of the hidden AST nodes [187]. This approach relied on the LLM's ability to infer the context surrounding the masked code to generate syntactically valid, realistic mutants. Hamidi et al. proposed an intent-based mutation technique that used a mask-and-predict approach on natural language intents (i.e., written descriptions of the code's purpose) for the method under test, and then requested an LLM to generate the methods described by the mutated intents [72]. This approach reflects method-level misunderstandings in the developer-implemented method. While the mask-and-predict style is popular and reduces resource

requirements, it only enables the language model to replace existing text and offers little scope for addition- or deletion-based mutation.

On the other hand, MUTAHUNTER, as discussed in Section 2.5.5, prompts the LLM to apply a mutant to a single line of code, with suggestions of the types of defects it may introduce [36]. This approach is not the subject of existing research studies, so it is unclear how it performs compared to other techniques. The BUGFARM approach first identifies hard-to-detect program statements through attention analysis, then asks LLM to mutate each hard-to-detect statement to form a single mutant [85]. While the overall mutant is constructed at the method level, mutations are applied to statements individually. BUGFARM mutants were harder to detect and harder to repair than both μBERT and LEAM mutants [85]. These techniques widen the scope of code mutation; however, they still constrain individual mutations to small code regions such as lines and statements.

Wang et al. conducted a comparative study of rule-based, deep-learning and LLM-based mutation (using multiple LLMs), considering many of the above approaches along with their own prompt [198]. LLM-based approaches proved more effective than rule-based approaches at coupling with real faults and detecting them. However, as has been a consistent problem in the field, rule-based mutants outperformed LLM-based mutants in terms of compilability, duplicate mutants, equivalent mutants and time cost. These factors remain limiting factors for LLM-based mutation and require further work to reduce them.

Equivalent Mutant Detection

The equivalent mutant problem is undecidable and often requires human analysis to determine whether a mutant is killable or semantically identical to the original SUT [29]. Tian et al. applied LLMs to the equivalent-mutant problem, finding that they outperform existing techniques, including Trivial Compiler Equivalence [107, 150, 185]. Applying LLMs to such coding tasks that require semantic understanding proves successful, and therefore, tasks like equivalent mutant detection stand to benefit.

Test Generation

Test generation is one technique used in mutation testing to improve test suite adequacy. Given a set of mutants, automated test generation techniques aim to achieve mutation adequacy (i.e., kill all the mutants). The LLM-based techniques often use rule-based mutants to prompt LLM-based test generation. For example, PRIMG uses a mutant-subsumption-trained model to predict the usefulness of surviving mutants, then prompts an LLM to iteratively design test cases targeting those mutants [26]. Similarly, MUTAP utilises surviving mutants from MUTPY to inform a prompt-based learning technique that

iteratively refines and generates Python test cases to kill the mutants [40]. Furthermore, Harman et al. use text descriptions of privacy concerns to prompt an LLM to generate mutants, then similarly use the LLM to generate killing tests [75]. Finally, Straubinger et al. employ a scientific debugging cycle, using the LLM to formulate hypotheses and experiments to explain mutant behaviour and, therefore, generate an appropriate killing test [181]. While high in cost, this technique outperformed Pynguin — a state-of-the-art Python test generation tool. As such, in some cases LLMs have been shown to be more effective than existing tools due to their context-awareness, enabling targeted testing rather than the search-based approaches used by techniques such as Pynguin [122, 181]. However, often LLMs perform worse than other tools, such as EvoSuite, due to hallucinations resulting in uncompilale test cases, leaving potential room for combined techniques [59, 60].

2.5.5 Implementations and tools

There are many mutation tools, implementations and prototypes available for both research and practical use, many of which are open source [170]. Given the rise of LLMs in recent years, the number of LLM-based prototypes is increasing; however, there are few well-maintained practical tools available. This section discusses both rule-based and LLM-based techniques and implementations, with a particular focus on Java, but noting where transferable techniques are implemented in other languages.

Rule-based Implementations and Tools

Rule-based mutation tools use pre-defined operators (see Section 2.5.1) to describe the transformation patterns that form mutants in the SUT. Due to language-specific nuances and differences, rule-based operators often need to be adapted, leading to multiple tools and implementations. For example, a statement in Java is defined differently from a statement in Python, and, as a result, the implementations and operator sets must differ [49, 51]. Many tools, such as MUJAVA for Java [124], MUTPY for Python [51] and HUMBUG for PHP [57], are no longer maintained and may therefore not work or be effective on recent versions of their respective programming languages. Popular tools include STRYKERJS for JavaScript [4], MUTMUT for Python [81], MUTANT for Ruby [173] and PIT for Java [37], which are actively maintained by open-source communities. This section goes into further detail on Java mutation testing tools and their deletion operators to provide background for Chapters 3 and 4.

muJava MUJAVA is an early automated Java mutation tool concentrated on object-oriented language mutation operators [124]. Object-oriented programming introduced a range of fault types that existing mutation techniques could not fully exploit. Class

mutation operators, therefore, need to be able to alter program structure, for example, changing access modifiers and deleting explicit calls to parent constructors [124]. Because existing techniques are unsuitable, MUJAVA uses bytecode transformation for structural mutants via *Mutant Schemata Generation* (MSG) [191]. To apply one-op mutation testing (see Section 2.5.3), Deng et al. implemented their definition of the SDL operator into the MUJAVA mutation tool [49, 123]. This implementation includes deletions of entire statement structures, from single-line statements to entire *for* statements. It also includes a basic set of default values for *return* statements. This involves returning the value "0" for *int*, *char*, *double*, *float*, *long* and *short*, as well as both "true" and "false" for *boolean* types. For the *String* type, MUJAVA returns *null*. MUJAVA is unmaintained and does not work with modern versions of Java, and is therefore seldom used in recent empirical studies.

Major MAJOR is a Java mutation framework that aims to reduce the overheads of mutation testing using a compiler-integrated mutator for Java and a mutation analyser [96]. Using compiler-based mutation avoids the need to recompile each mutant at run time. One of MAJOR's unique features at release was state infection to reduce ineffective execution. Under weak mutation, state infection (the second stage of RIPR in Section 2.2.2) kills a mutant, whereas strong mutation requires a propagation to output and to be revealed by, for example, an assertion. The system performs an initial test suite run to identify which test executes each mutant, and monitors state infection to compare states before and after a mutant in the weak mutation run. While Major is useable in both research and practice, Sanchez et al. found that 42 of the 6307 repos investigated used Major, suggesting it does not have a large share of the mutation field compared to, for example, PIT [37], which was used by 1340 of the repos [170]. Just et al. identified statement deletion as akin to a genuine fault, yet requiring a stronger mutation operator [98]. Using MAJOR's statement deletion operator implementation, Just et al. found that deleting control flow statements can lead to uninitialised variables or unreachable code errors. This is due to the challenges of implementing such a tool for Java. MAJOR is not open source and, as such, cannot be extended for experimental research by anyone other than the tool owners.

PIT PIT is a state-of-the-art mutation testing tool that inserts and executes mutants in Java programs by using Java bytecode manipulation designed for use on real-world code bases [37]. PIT can be used via a command-line interface or Ant or Maven, making it practical for developers to integrate it into their current processes. It is also efficient, as it runs only tests that execute the location of each mutant, significantly reducing run time. PIT cannot be used for Java statement deletion because, due to bytecode manipulation, its operations do not directly map to the program's source code, making it impossible

to accurately represent all source code statements. Since its release in 2016, PIT has been maintained in an open-source GitHub repository and has received regular fixes and updates. Researchers have also built multiple tools and extensions on top of PIT due to its recognition as a core tool in the field, including Descartes, an extreme mutation engine [197].

Descartes for PIT The DESCARTES mutation engine replaces the GREGOR engine in PIT to implement XMT for finding pseudo-tested Java methods [137, 195]. Compared to GREGOR, DESCARTES is efficient, as fewer mutants are generated under XMT, and does not include any further operators. At the Java bytecode level, it is easy for a tool to remove a method, and for a developer to translate it back to the method body the tool deleted. Therefore, it is appropriate to implement XMT by manipulating a Java program's bytecode.

Reneri While not a mutant-producing tool, RENERI observes pseudo-tested method executions from PIT using the DESCARTES mutation engine, both with and without a method body, enabling it to suggest potential solutions for pseudo-testedness, such as adding tests and verifying variable values [196].

LLM-based Implementations and Tools

As overviewed in Section 2.5.4, recent advancements in language models, particularly LLMs, have driven the rise of model-based mutation testing tools. Initial tools such as DEEPMUTATION and SEMSEED were trained on a dataset of real faults [82, 157, 168, 184, 188]. Yet, for practical reasons, more recent tools use general-purpose LLMs, as discussed in Section 2.5.4. This section continues by discussing the two LLM-based mutation techniques, LLMORPHEUS and MUTAHUNTER, that are empirically evaluated in Chapter 5.

LLMorpheus LLMORPHEUS is a research prototype mask-and-predict style LLM-based mutation tool that targets pre-specified AST node types, with a prompting strategy that requires pairing with a language-specific AST parser [187]. By using pre-determined AST node types for mutation, LLMORPHEUS functions similarly to rule-based mutation techniques, in which mutation locations are set by pre-defined rules. First, LLMORPHEUS parses the AST, identifying the locations of (1) `if`, `while`, `do-while` and `switch` condition statements; (2) `loop` initializers, updaters and full headers and (3) `function` call receiver arguments and argument sequences. Each of these locations is then located and one-by-one replaced with "`<PLACEHOLDER>`" in the source code fragment and entered into the prompt. The system prompt gives a brief overview of mutation testing before the user prompt presents the LLM with the source-code fragment, both with and without

the "<PLACEHOLDER>". The prompt then suggests changes such as operator modification, variable reference switching and method, function, or object property references and requests that three possible mutations be output. In the original tool, LLMorpheus uses a modified version of StrykerJS — a state-of-the-art JavaScript mutation tool — to execute and evaluate the generated mutants; however, it has since been adapted to other languages [198]. Although operating similarly to a rule-based approach using preset locations, the LLMorpheus mutations can be diverse and context-specific, as the LLM suggests three separate mutations for each pre-specified node rather than a single preset rule-based transformation. As such, LLMorpheus could generate mutants that resemble real bugs that could not be produced by StrykerJS.

Mutahunter Mutahunter is an open-source, LLM-based tool capable of mutating code in any language supported by an LLM [36]. Its system prompt provides an overview of mutation testing and outlines guidelines for potential changes, including syntax-based logic modifications, return type alterations, failure simulation and data-handling errors. The user prompt directs Mutahunter to target function blocks and "critical areas" while constraining mutants to a single line only. Unlike LLMorpheus, Mutahunter does not "mask" any code but instead prompts the LLM to modify existing lines directly, providing only the original code and, optionally, the AST for context.

Both LLMorpheus and Mutahunter constrain the LLM to mutating small areas of code, namely AST nodes and single lines, respectively. While this ensures the LLM is constrained, these techniques do not give the LLM the freedom to produce larger mutants.

2.5.6 Mutation Testing in Practice

As with most research, the question of how mutation testing can be used in practice is critical yet under-explored, often due to the impracticalities of conducting studies in an industrial context. Research regularly changes mutant operator sets and best practices, and therefore, research techniques and comparisons, particularly of mutation scores, may not be useful for helping developers improve their test suites. To a developer, the mutation score provides no actionable content without the mutants, and questions can be raised about how useful it is to combat every individual mutant. To management, decisions on which mutant sets developers should use should be updated regularly to reflect new research. This is where a tool like PIT, which offers a traditional set of mutants, can be used [37]. Large companies such as Google and Facebook have shown how mutation testing can be applied in an industrial context.

Google

Google is a prominent use case of mutation testing in industry, with ongoing research to make it actionable for developers and feasible at large scale [159]. Petrović et al. share Google's approaches to integrating mutation testing into large code bases and developer workflows by recognising that a developer's unit of work is the commit and scaling mutation testing accordingly [161]. Furthermore, ensuring the mutants presented to developers are (a) productive and (b) reasonable is essential to ensure developers engage with and benefit from mutation testing [159, 161]. This includes removing arid nodes (see Section 2.5.3), reducing the number of mutants shown to a developer and ensuring the mutants shown are relevant to the current commit, increasing the chances of them being acted on [159]. Furthermore, using mutants to improve the test suite is a more reasonable ask of a developer than requiring the test suite to be mutation adequate [163]. Removing unproductive mutants and enabling developers to push back with reasons why a mutant should not be addressed increases developer engagement and leads to improved test suites and source code refactoring [162]. Overall, the research at Google has demonstrated that mutation testing is usable and feasible at scale; however, it must be adapted and integrated into a developer's workflow, presenting only the most relevant and productive mutants [160].

Meta

At Meta and their subsidiaries, mutation testing is less integrated into the code review process than at Google. The "Mutation Monkey" system at Facebook that learned "error-inducing" patterns from Defects4J collection of reproducible bugs and Facebook-specific defects [21]. The learned operators had a significantly higher survival rate than generic traditional mutation operators (i.e., 60–70% vs 10–15%), suggesting learned and, therefore, likely more realistic operators make mutation testing more efficient, particularly at scale, as computational resources are not wasted on generating and executing trivial mutants [21]. More recently, Meta's agentic LLM-based ACH (Automated Compliance Hardener) tool generates mutants and test cases specific to privacy issues in code [75]. ACH takes free-form text from sources such as existing development faults, user requirements, constraints, concerns and regulations to "harden" the test suite against such compliance concerns. It then prompts an agent to generate mutants based on existing code and test cases, with a further LLM-as-a-judge agent to evaluate mutants for equivalence. The surviving, non-equivalent mutants then guide the test generation process. Empirical results found that if code coverage alone had been used as a test generation criterion in this scenario, 48.5% of the additional mutant killing tests would have been discarded, showing the importance of mutation testing for test suite improvement [75].

XMT in practice

Betka et al. compare the computational cost of traditional mutation testing with that of extreme mutation testing (XMT) in practice [22]. The industrial case study comprised a large software project with over 11,000 unit tests, culminating in a discussion with two developers about 25 specific pseudo-tested methods XMT had identified. XMT showed shorter execution and analysis times and identified larger testing gaps. Therefore, the developers were (a) more likely to use XMT and (b) more likely to act on its output. The mutants identified testing issues related to the API functionality, code review practices, developer responsibility and sensitive data processing [23]. Not only were issues code- and test-related, but they also raised discussion of which developer/team was responsible for addressing a given pseudo-tested method. These results suggested XMT should be integrated into existing coverage reports, due to being easy to understand with actionable outcomes [23].

Mutation Testing in Open-Source Projects

Sanchez et al. revealed the gap between practice and research through systematically searching GitHub for mutation tools and their uses [170]. Of the 127 tools they identified, 7 of the top 10 were not mentioned in any research papers and were used primarily for development. A study of the top 10 tools found that the use of mutation testing among developers is increasing, and that there has been a movement away from Java and C/C++ towards web development languages such as JavaScript in recent years. A survey of 104 open-source contributors found high satisfaction with mutation testing among developers, particularly for improving test suite quality, bug detection and code maintenance [171]. However, as is often the case, performance remains the limiting factor in adoption. This further calls for work on reducing costs in mutation testing.

Mutation Testing in the Safety-critical industries

Safety-critical industries often use structural coverage metrics, such as MC/DC, to measure test adequacy, which are often stipulated by the industry safety standards [33]. Delgado-Pérez et al. performed an industrial case study on safety-critical C code from the UK nuclear industry to evaluate the fault-finding effectiveness of test suites that already adhere to industry structural coverage standards [46]. Empirical results showed the existing test suite killed 92% of non-equivalent/non-duplicate mutants, yet improving the test suite to achieve 100% mutation adequacy required increasing the number of tests by 18%. As such, mutation testing may be worth the overhead for ensuring test suite quality in safety-critical systems [46].

2.5.7 Hazard Analysis in Mutation Testing

Chapter 5 employs hazard analysis techniques to generate LLM-based mutants. To contextualise these techniques, this section outlines the relevant frameworks and how applying them to mutation testing can systematically generate synthetic faults.

Hazard analysis evaluates the potential risks involved in a project or task by a step-by-step review of operating procedures [55]. Particularly critical in high-risk industries such as chemical and manufacturing, hazard analysis enables engineers to identify the potential causes and consequences of unintended behaviour. Depending on the process under scrutiny or industry regulations, safety engineers use established frameworks such as Hazard and Operability Study (HAZOP) and Systems Theoretic Process Analysis (STPA) [108, 119].

HAZOP (Hazard and Operability Study)

HAZOP (Hazard and Operability Study) is a hazard analysis procedure for identifying potential deviations from the intended design [53, 108]. Given a set of design intents, HAZOP analysis involves applying a prescribed set of guide words to each intent to identify potential deviations. The core guide words are as follows, but can be adapted depending on the scenario:

1. **No or Not**: A negation of the original intent.
2. **More**: An increase in parameters of the original intent.
3. **Less**: A decrease in parameters of the original intent.
4. **Part of**: Restricted implementation of the original intent.
5. **Reverse**: Logical opposite of the original intent.
6. **Other than**: Alternative implementation of the original intent.
7. **As well as**: More intents than should be present.

Each of these guide words manipulates a design intent, potentially leading to unintended deviations or behaviours. The experts then evaluate this behaviour to understand the potential causes, consequences and hazards of the deviation, and therefore ensure the relevant protective systems are in place to prevent the causes of the deviation. Finally, the experts evaluate the remaining risk given the available protective systems. Manual HAZOP evaluation in process engineering has been noted to be a time-consuming exercise and highly dependent on the expert knowledge [53, 156].

Conceptually, the HAZOP guide words function similarly to mutation operators, creating potential faults in the system under scrutiny. As such, HAZOP has been applied

to the Java programming language to form a set of "rigorous" yet plausible rule-based operators [105]. Enforcing the meticulous nature of HAZOP ensures that, if addressed, a HAZOP-based mutation-adequate test suite will cover all language constructs. Hazard-based mutation has also been applied to use-case modelling to systematically evaluate use-case models, create a comprehensive set of mutation operators for Restricted Use Case Models (RUCM) and select traditional mutation operators for safety-critical systems [71, 206].

STPA (Systems-Theoretic Process Analysis)

In safety engineering, STPA (System-Theoretic Process Analysis) identifies accidents as a control problem [119]. That is, hazardous control actions, whether performed by humans or software, lead to failures. For instance, given a train with correctly functioning accelerator and passenger doors, if the driver (i.e., the controller) opened the doors while travelling at maximum speed, this would be an unsafe control action (UCA). STPA enables engineers to identify and scrutinise potential UCAs, ensuring they are mitigated and their risks documented. The scenarios considered in STPA are:

1. A required control action is *not provided*.
2. An unsafe control action *is provided*.
3. A required control action is provided *too early* or *too late* (i.e., incorrect timing).
4. A required control action is *stopped too soon* or *applied for too long* (i.e., incorrect duration).

STPA models each action a controller may take, before applying the guide words and envisaging the UCAs. While not currently applied in mutation testing, the STPA guide words also reflect mutations of the expected control behaviours and, as such, could be applied to software control flow.

LLMs in Hazard Analysis

All methods require a team of experts who can analyse and critique the process design. As such, hazard analysis can be time-consuming and subjective. To reduce the time required of experts, safety engineers have applied LLMs to the task of hazard analysis [52, 117]. However, the common finding is that in safety-critical scenarios, LLMs lack the expert knowledge to perform hazard analysis unsupervised and, as such, can serve as supportive tools, ideal for rapidly generating candidate deviations and UCAs, rather than expert replacements [32, 52, 102, 117]. Diemert et al. also found that incorrect LLM interpretations of the specification were useful, as they highlighted specification gaps and weaknesses that had not previously been considered [52]. However, LLMs have not yet

applied hazard analysis techniques for generating mutants for program code as described in Chapter 5.

2.6 Conclusion

This chapter evaluates the literature surrounding the research topics covered in the thesis. In doing so, it provides both the necessary background and relevant related work to provide context and motivation for the approaches proposed and evaluated in subsequent chapters. Firstly, Section 2.2 provided an overview of the role of unit testing in software testing and the criteria for fault detection. This section then emphasised the limitations of code coverage criteria: they do not require assertions and cannot guarantee that tests that meet them will reveal faults. Chapter 3 builds on this to demonstrate how statements fulfilling code coverage alone can be deletable without changing test outcomes, and therefore require an assertion to ensure the statement behaviour is as expected. Next, Section 2.4 demonstrates how oracle-based adequacy criteria can address the problem of revealability, yet implementations remain research prototypes and unutilised due to limited knowledge. The relationships among these criteria are empirically evaluated in Chapter 4 to ensure that developers can make informed decisions about their purpose. This chapter concludes with an overview of mutation testing in Section 2.5 and highlights the challenges relevant to this thesis. Mutation testing is a costly endeavour, and as such, much effort is devoted to reducing its cost and increasing its effectiveness. Applying LLMs to the challenge has enabled context-aware mutation; as of yet, there is not a systematic way to approach this. Chapter 5 therefore combines LLM-based mutation from Section 2.5.4 and Hazard Analysis techniques from Section 2.5.7 to guide method-level mutation.

Chapter 3

Empirically Evaluating Pseudo-testedness at the Statement Level

The contents of this chapter are based on the paper "M. Maton, G. M. Kapfhammer, and P. McMinn. Exploring Pseudo-Testedness: Empirically Evaluating Extreme Mutation Testing at the Statement Level. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2024" ().

Author Contribution For this publication, I was responsible for extending and repurposing an existing prototype created by the third author to integrate and extend statement deletion to detect pseudo-tested statements, under the guidance and supervision of the second and third authors. I designed and implemented the experimental design, collected data and drafted the paper, incorporating the second and third authors' feedback and text edits for the final version.

Use in this Chapter For this chapter, the content of the original paper has been adapted and edited to align with the background provided in Chapter 2 and to define the terminology used in Chapter 4. To maintain a consistent voice throughout this thesis, I have independently re-authored all sections drafted by co-authors.

3.1 Introduction

Extreme mutation testing (XMT) is an approach for detecting deficiencies in a test suite that individually deletes method bodies in covered program code and observes whether the test suite detects their absence [137]. If each method body deletion is successfully detected — i.e., at least one test now *fails* — the method is said to be "required" for the test suite to pass [195]. However, if a deletion goes unnoticed — i.e., all the tests *still pass*

— the method is instead said to be "pseudo-tested". That is, a pseudo-tested method is a program method executed by a test suite whose behaviour is not checked by a test's assertions, either directly or indirectly. Since pseudo-tested methods indicate apparent deficiencies in a program's test suite, developers can use such knowledge to strengthen the program's tests, removing weaknesses related to pseudo-testedness and ensuring that each method's behaviours are effectively tested [137].

Pseudo-tested methods have been shown to be regions of lower mutation score than required methods, further showing how they can be "blind spots" for a test suite [195]. As areas of the code that can be completely removed without impacting test outcomes, it is intuitive and self-fulfilling that synthetic bugs are also less likely to be detected by the test suite. Tools such as Descartes [197] and Reneri [196] can identify and target pseudo-tested methods without necessitating full mutation testing, while also suggesting improvements to developers. The quick feedback XMT provides enables developers to address issues as their code evolves. However, XMT's use of a coarse-grained transformation (i.e., method body deletion) may overlook subtler instances of pseudo-testedness. Methods classified as required may not be thoroughly tested, leaving the test suite to miss some issues that traditional mutation testing may detect. That is, a technique is needed to help developers quickly find smaller units of pseudo-testedness, in addition to complete methods, without resorting to full, costly mutation testing.

This chapter demonstrates that the statement deletion (SDL) mutation operator [47] can be used to identify pseudo-testedness at the level of individual program statements and statement blocks (**R1**). The SDL operator is a versatile mutation operator, capable of mutating most of the code that can appear in a program [190]. Furthermore, Deng et al. [49] found that a test suite that achieves a 100% mutation score using only the SDL operator achieves an average of a 92% mutation score with a full set of operators, but with 81% fewer mutants. Meanwhile, developers at Google also use statement block removal to highlight potentially redundant code [159, 161]. However, SDL is often omitted or only partially implemented in widely used Java mutation tools such as PIT [37] and Major [96], and research has yet to evaluate SDL's usefulness beyond its role as a mutant-reduction technique [49, 43]. To ensure SDL is suitable for pseudo-tested statement identification, this chapter includes technical developments to SDL to include previously excluded statement types such as *break*, *continue* and *variable declaration*, as well as extend *return* statement mutation to include multiple mutant operators, implemented in our tool PseudoSweep (see Appendix A for implementation details).

The experiments in this chapter are the first to use SDL to identify statements that are pseudo-tested by a test suite, finding 722 examples of pseudo-tested statements in both four frequently-studied, large, Apache Commons Java projects, as well as 23 projects randomly selected from the Maven Central Repository [2], the largest official software repository for Java. Critically, we found that XMT misses 48% of the pseudo-tested state-

ments (**A1**), as they appear in methods it classifies as required, an oversight that may lead developers to an incorrect assumption that they have addressed all pseudo-testedness — and delaying the detection of testing deficiencies related to these statements to traditional, resource-intensive, mutation testing [101]. We further found that pseudo-tested statements achieved lower mutation scores (0.57) than required statements (0.80) and that traditional mutation testing does not typically mutate some Java statement types, such as *continue*, *break* and *throw* statements (**A2**), leaving gaps in test suite evaluation [99, 98]. Finally, we manually analysed a sample of 119 pseudo-tested statements to understand their causes, revealing that most are due to specific test suite inadequacies (e.g., missing tests or partial assertions) that XMT and traditional mutation testing cannot highlight because their operators are inapplicable (**A3**). The contributions of this chapter, then, are as follows:

A1 A quantitative evaluation of the frequency of pseudo-testedness at the statement level, specifically those left undetected by XMT (Section 3.5.2).

A2 A quantification of the differences between pseudo-tested statements and other covered statements using traditional mutation scores (Sections 3.5.3 and 3.5.4).

A3 A qualitative manual analysis of the causes of pseudo-tested statements to understand the test suite inadequacies that they uncover (Section 3.5.5).

3.2 Defining a pseudo-tested statement

Where the test suite executes a production code element (e.g., statement or method), yet the same element can be deleted from the project without altering the test outcomes, this code is said to be "pseudo-tested" [129, 137]. By adapting the definitions first defined in extreme mutation testing, as described in Section 2.5.3, pseudo-tested statements can be categorised as follows:

Definition 3.2.1 (Pseudo-tested Statement)**,** Code that the tests execute, but is removable without changing test outcomes.

Definition 3.2.2 (Required Statement)**,** Code that the test suite executes but is *not* removable without changing test outcomes.

For this chapter's evaluation, pseudo-tested elements may include partially-tested elements (see Definition 2.5.9), as both are potential areas of action for a developer.

3.3 Approach

Our automated approach finds relevant pseudo-tested statements by extending the SDL operator to include further transformations and implementing a supplemental approach

PseudoSweep Classifications
□ Required Statements ■ Pseudo-tested Statements

```
1  /**
2   * Decode bytes encoded with Percent-Encoding based
3   * on RFC 3986. The reverse process is performed in
4   * order to decode the encoded characters to Unicode.
5   */
6  @Override
7  public byte[] decode(final byte[] bytes)
8                           throws DecoderException {
9    if (bytes == null) {
10     return null;
11   }
12   /* ... */
13   try {
14     /* ... */
15     buffer.put((byte) ((u << 4) + l));
16   } catch (final ArrayIndexOutOfBoundsException e) {
17     throw new DecoderException(/* ... */);
18   }
19   /* ... */
20   return buffer.array();
21 }
22
23
24 ✓@Test
25   void testPercentEncoderDecoderWithNullOrEmptyInput()
26                                     throws Exception {
27    /* ... */
28    assertNull(percentCodec.decode(null),/* ... */);
29    /* ... */
30  }
31
32 ✓@Test
33   void testDecodeInvalidEncodedResultDecoding()
34                                     throws Exception {
35    /*...*/
36    try {
37      percentCodec.decode(/* Invalid Encoded Result */);
38    } catch (final Exception e) {
39      assertTrue(
40          DecoderException.class.isInstance(e) &&
41              ArrayIndexOutOfBoundsException.class
42                                  .isInstance(e.getCause()));
43    }
44  }
```

Listing 3.1: An example of a "required" method containing a "pseudo-tested" statement from the *org.apache.commons.codec.net.PercentCodec decode* method, and two unit tests, including *testDecodeInvalidEncodedResultDecoding*, from the test class *org.apache.commons.codec.net.PercentCodecTest* (© 2024 IEEE).

to identify pseudo-tested methods using the definitions in Section 2.5.3. Since identifying pseudo-tested statements within pseudo-tested methods does not provide any new information to a developer, it is more helpful to explore pseudo-tested statements external to pseudo-tested methods, that is, those that XMT would not reveal. This is because where a method is pseudo-tested, it has already been labelled as an area of testing weakness and, therefore, to identify each tiny problem within the larger problem is both repetitive to a developer, and does not provide additional guidance for improving their test suite.

For example, for the *decode* method in Listing 3.1, identifying pseudo-tested methods alone would declare this method as "required" for the test suite to pass. However, using coverage and SDL information, a tool can remove the *throw* on Line 17 without causing any test cases to fail. We show the *testDecodeInvalidEncodedResultDecoding* test case as

it targets this line. However, despite covering this statement and using an assertion to check the exception type, the structure of the test means that if it does not trigger the program code to throw an exception in the first place, the test does not reach the *catch* statement containing the assertion, meaning the test does not fail. With this information, a developer could add the following JUnit fail assertion:

```
1  fail("Expected exception was not thrown");
```

to the test between lines 37 and 38 to fail when the test has reached this point without catching an exception, ensuring that future maintenance does not accidentally introduce this fault of omission. We implemented this approach in a tool called PseudoSweep, as explained in the remainder of this section. For further information on the PseudoSweep tool itself, please refer to Appendix A.

3.3.1 Method Classification

If PseudoSweep removes a covered method without causing the test suite to fail, the method is pseudo-tested. Where a method requires a *return* statement for compilation, PseudoSweep uses a pre-defined set of default values as used in the current implementation of Descartes [195]. For example, where a method returns an integer value, PseudoSweep replaces the method body with *return 1* and then tries again with *return 0*. By checking two common default return values (0 and 1) for each method, PseudoSweep avoids returning the method's existing return values and, therefore, the value that the tests expect. For a complete description of the operators, please refer to Appendix Section A.2.1. By first classifying each method as not covered, required, or pseudo-tested (where the pseudo-tested subset includes partially-tested methods), we can identify the immediate issues of pseudo-testedness, like other XMT tools.

3.3.2 Statement Classification

Next, PseudoSweep locates the pseudo-tested statements and uses the method classifications to identify pseudo-tested statements within the required methods. Note that non-covered methods cannot contain pseudo-tested statements, as a pseudo-tested statement must by definition be an executed statement.

Basing the core deletions on the statement deletion (SDL) operator [49] would mean that PseudoSweep could only evaluate statements for pseudo-testedness that it can delete in their entirety without causing compilation issues or by replacing return statements with a single value per type. For example, for integer *return* statements, the SDL operator returns only `0`, and, therefore, if the statement already returned `0` this value would be identical and expected by the test suite. Therefore, we extended SDL to enable PseudoSweep to delete variable declarations, *lambda* statements and labelled loops, as explained in Section 3.3.3.

We also found that SDL and XMT would conflict when classifying some single-statement methods. Given the following method, SDL classifies the *return* statement on Line 2 as "required" because it applies only a single killable mutant of `0`.

```
1  public int subtract(int a, int b) {
2      return a - b;
3  }
4
5  @Test
6  public void testSubtractEqualValues(){
7      int result = subtract(2, 2);
8      assertEqual(result, 0);
9  }
```

Yet, XMT would apply two mutants, `0` and `1`, where only one (`return 1`) is killable, and classify the method as pseudo-tested. Therefore, we have implemented SDL with the same default operators (i.e., combining SDL with a Return Value Mutation operator), as in XMT in Descartes Engine, to improve its effectiveness [197].

3.3.3 Metamutant

PseudoSweep implements statement deletion and method deletion by instrumenting source code to create a metamutant [191]. This metamutant ensures that all statement and method deletions are always compilable. To create the metamutant, PseudoSweep inserts *if* statements around each statement/method body in the source code, enabling it to execute each element conditionally [100]. The condition of the inserted *if* statements calls PseudoSweep to check if it should run the contained element. The following Java code gives a simplified example of how the tool instruments most program statements.

```
1  if (PseudoSweep.exec(ID)){
2      i++;
3  }
```

The full instrumentation also includes element type and class information, particularly as the element type dictates how instrumentation is applied. When a scope change causes compilation issues, the tool uses the default values to enable deletion.

```
1  String s = "";
2  if(PseudoSweep.exec(ID)){
3      s = "actual";
4  }
```

In the example above of a variable declaration metamutant, PseudoSweep enables compilation by assigning a default value and conditionally assigning the original initialisation.

3.3.4 Evaluating Deletions

We can use the previously described metamutant to instrument each element. PseudoSweep runs each deletion with a timeout relative to the original test case runtime to halt any infinite loops introduced by the metamutant. The tool then sequentially "removes" each element and executes the test. If the test passes, the tool runs it twice again to check for flakiness [154]. We skip flaky tests because they may affect the reliability of

pseudo-testedness detection. The results are then internally analysed to identify elements whose removal does not cause test failures, and PseudoSweep presents these to the user as pseudo-tested elements.

3.4 Evaluation

Since prior work suggests that pseudo-tested methods exist in all projects [137, 195, 23], we first investigate how common pseudo-tested statements are within our project set and their relationship with the "required" methods (RQ1). Next, we determine whether mutation scores for pseudo-tested statements are lower than for required statements (RQ2), as occurs at the method level [195, 23]. Using mutation scores calculated by PIT (a state-of-the-art Java mutation tool), we explore whether traditional mutation testing effectively targets pseudo-tested statements (RQ3). Finally, since prior work showed that XMT overlooked concerning issues in required methods [23], we use statement deletion to identify issues missed by XMT and compare them with the findings of traditional mutation testing, therefore highlighting the gaps between the techniques (RQ4). Ultimately, this chapter answers the following four research questions:

RQ1: How frequent are pseudo-tested elements?

RQ2: Do pseudo-tested elements have low mutation scores?

RQ3: Does PIT's default set of operators effectively highlight deficient testing with respect to pseudo-tested statements?

RQ4: What are the causes of pseudo-tested statements?

3.4.1 Projects

This chapter investigates pseudo-testedness in diverse, real-world Java projects. Table 3.1 shows that we used four large open-source Apache Commons projects and 23 open-source projects randomly chosen from the Maven Central Repository [2]. When randomly selecting the Maven projects, we adopted the following criteria: uses JDK versions 8–11 (due to tooling restrictions from using JavaParser [178]); single module (currently PseudoSweep only analyses single-module projects); explicit type declarations; uses JUnit 4 or 5; one or more tests that compile and pass before using PseudoSweep.

To ensure that we studied a diverse set of real-world projects of different sizes and test suite maturities, we followed the project sampling method used by Gruber et al. to identify over 38,000 Maven projects [70]. As the prohibitive time costs of running PIT prevented the use of all these projects, we randomly sampled 180 of them from this list, collected their source code from GitHub, and retained only those that met our inclusion

Table 3.1: Details for each evaluation project, including its GitHub owner and repository name and the numbers of methods (# METHOD), statements (# STMT), tests (# TEST), assertions (# ASSERT), and PIT mutants (# MUTANT), and JaCoCo's Bytecode Instruction Coverage (% BCOV) and Line Coverage (% LCOV) and PseudoSweep's Statement Coverage (%SCOV). The names of the Apache Commons projects are bold while those of the Maven projects are not. The portion of a project name that is in italics is used as the project reference for the remainder of the paper. The "–" symbol indicates that it is a non-meaningful total ().

Owner / *project*	# METHOD	# STMT	# TEST	# ASSERT	# MUTANT	% BCOV	% LCOV	% SCOV
LindenY / *asw4j*	170	816	42	215	504	48	49	49
authlete / *authlete-jose*	60	570	15	15	324	49	55	53
IvoNet / *beanunit*	44	275	44	56	183	80	75	68
sigpwned / *chardet4j*	80	682	13	13	732	91	68	62
abedra / *chronometrophobia*	65	85	3	5	238	41	55	52
apache / *commons-cli*	140	745	448	688	753	96	96	97
apache / *commons-codec*	625	3933	923	2536	4060	97	95	90
apache / *commons-csv*	145	960	468	1445	718	98	99	99
apache / *commons-math*	6467	46206	6136	12402	45646	92	90	87
coodoo / *coodoo-listing*	81	739	64	84	477	13	16	13
kestreldigital / *data-conjuror*	12	29	2	3	26	54	59	59
fuinorg / *ext4logback*	7	45	4	16	26	51	59	44
vebqa / *f3270*	204	1299	7	6	864	4	4	0.8
mervinkid / *jargser*	10	96	1	6	86	88	84	86
erdtman / java-*json-canonicalization*	65	1102	5	1	993	49	40	33
clerezza / *jena.serializer*	2	32	5	7	17	88	89	88
rbkmoney / *kafka-common-lib*	27	117	6	10	73	29	32	42
Naoghuman / *lib-logger*	36	95	3	0	43	25	23	19
harium / *marine*	14	45	2	2	31	6	7	4
awslabs / *payload-off-loading*-java-common-lib-for-aws	37	188	40	92	100	71	71	68
premiumminds / *pmpersistence*utils	27	228	19	24	106	66	71	68
sergiodeveloper / *sequencepattern*	114	493	16	16	397	41	48	41
andreidore / *smartbill*-java-*client*	16	200	11	9	58	19	21	23
tblsoft / *solr-cmd-utils*	807	7280	179	449	3204	24	24	23
bordertech / *wcomponents*-sass-*compiler*	7	62	1	2	28	70	74	74
wildflyswarm / *wildfly-swarm-spi*	64	268	12	156	180	47	46	40
vincentzurczak / *xml-region-analyzer*	10	222	9	273	196	96	97	97
Total for Apache Commons Projects	7377	51844	7975	17071	51177	–	–	–
Total for Randomly Selected Projects	1959	14968	503	1460	8886	–	–	–
Total for All Projects	9336	66812	8478	18531	60063	–	–	–

criteria, yielding a set of 23 projects. We also included four commonly used Apache Commons projects to connect our results about pseudo-tested statements to prior studies on pseudo-tested methods [137, 195].

3.4.2 Tooling

Since it is the state-of-the-art mutation testing tool for Java, we used PIT version 1.15.6 with JUnit5 plugin version 1.2.1 and the Gregor mutation engine to calculate the traditional mutation scores [37]. We used PseudoSweep, as described in Section 3.3, to analyse pseudo-tested statements and methods in the chosen projects. We used PseudoSweep's implementation of both the SDL mutation operator and XMT because the existing tools (e.g., muJava, Major and PIT with the Descartes engine) were unsuitable for the following reasons.

For SDL, we could not use muJava [49, 123, 124, 125] because it only supports up to Java 1.6 and was thus not applicable to recent Java projects. Moreover, Major's implementation of SDL only incorporates a subset of the statement deletion operator trans-

formations [96]. MAJOR's source code is unavailable, and therefore, we could not extend the operator set. Statement deletion is also less useful when implemented at the bytecode level. Since Java bytecode does not directly translate into Java statements, extending PIT's GREGOR Engine to support SDL may not always yield suitable statement-deletion mutants. Finally, the most recent version 1.3.2 of the DESCARTES engine [197] for performing XMT with PIT restricts it to an old 1.7.0 version, thereby limiting compatibility to an out-of-date JUnit5 plugin version 0.16.

3.4.3 Method

We took the following steps to set up each of the chosen projects. Since every project used Maven, we configured each project using its "Project Object Model" (POM) file. For each project, we duplicated the POM file. We then added PSEUDOSWEEP as a dependency to one, and PIT with the Gregor Engine as a plugin to the other, therefore ensuring their configurations did not interfere with each other. We used a script to execute both tools and direct each tool's output to a separate directory. As shown in Table 3.1, we also collected the JaCoCo coverage scores to characterise a project's level of testedness. Finally, we implemented and ran scripts to characterise the projects based on the mutants generated by XMT, SDL and PIT. This enabled us to report metrics, such as the number of pseudo-tested statements within required methods and the types of statements that contained the most PIT mutants. We used the data generated by these scripts to answer RQ1 to RQ3.

We adopted two approaches to studying pseudo-tested statements. First, we calculated the total number of pseudo-tested statements (P). We also investigated the subset of P that occurs in required methods (PiR). Any pseudo-tested methods identified by extreme mutation testing would already highlight that pseudo-testedness is present, meaning that identifying pseudo-tested statements in pseudo-tested methods does not reveal new information to a developer and is thus not a focus of this chapter. However, since extreme mutation testing can only reveal pseudo-tested methods and cannot reveal individual pseudo-tested statements within the required methods (PiR), this chapter studies the frequency of PiR statements in the project set. It is worth noting that it is possible for required statements to appear in pseudo-tested methods, however this is a rare occurrence. This is generally due to sequences of statements being collectively deletable, yet deleting a single statement on its own will cause an exception to be raised or a test to fail.

To answer RQ4, we manually analysed a sample of pseudo-tested statements found in the required methods. We investigated these elements because extreme mutation testing would not highlight their potential issues. The first author manually analysed a sample of 119 pseudo-tested statements across 30 required methods to identify their causes. We pro-

duced the sample by ordering the required methods by the number of pseudo-tested statements, then selecting methods with varying counts of pseudo-tested statements (greater than 0) from different projects until we had a sample size of at least 100 pseudo-tested statements within required methods.

First, we manually identified and removed 12 "uninteresting" program statements that PseudoSweep mutated and ultimately labelled as pseudo-tested. Also called "arid nodes", these statements represent parts of a program's abstract syntax tree (AST) that perform tasks like logging and thus do not implement features that should be subjected to mutation testing [159]. Since the current implementation of our tool does not filter out these statements, they were a subset of the pseudo-tested statements in the required methods. The manual analysis process took considerable time, and therefore in future work, we will improve PseudoSweep to remove arid nodes from consideration automatically. Finally, we removed four equivalent mutants introduced by PseudoSweep when an inserted default value was equivalent to the expected value. We manually removed each statement one at a time to confirm that it was pseudo-tested before looking at the tests that covered the method to determine why removing the statement would not affect the test suite outcomes. While manually reproducing pseudo-testedness, we identified and removed 11 statements in total from the sample that PseudoSweep had misclassified as pseudo-tested due to incorrectly handling the JUnit annotations and threading. To answer RQ4, we then identified the causes of the remaining 92 (*119(PiR)−12(arid)−4(equivalent)−11(unhandled)=92*) pseudo-tested statements, ultimately grouping them into meaningfully labelled categories.

3.4.4 Threats to Validity

External Validity We studied 27 open-source projects, selecting four open-source Apache projects (i.e., *commons-cli*, *commons-codec*, *commons-csv* and *commons-math*) due to their widespread use in the testing literature and their comprehensive test suites [195, 197, 80, 67]. Due to expense of running PIT, we randomly selected 180 projects from the Maven Central Repository and identified the subset of 23 projects that met our project criteria. We then forked their source code from the most recent GitHub commit. We used these projects to study pseudo-testedness across projects of various sizes and test suite maturities. Since our findings may not generalise to other projects, further studies should extend the experiments in this chapter to a larger set of projects.

Internal Validity The PseudoSweep tool and the data analysis scripts are not exempt from defects. We extensively tested the tools, but instrumentation limitations may cause them to fail for code patterns we have not yet encountered. We further ran all of each project's tests three times and recorded execution times to check for flakiness [154].

If a test is flaky, we cannot rely on it to identify pseudo-tested elements [176], so we skip this test. We also ran each test three times against a mutant to mitigate potential flakiness. We leave an investigation of the impact of flakiness on pseudo-testedness detection for future work. We also checked the results, addressed unexpected values and created a replication package for the paper this chapter is based upon [128].

We based our categorisation of statement types on those in JavaParser and combined sub-categorisations as part of the instrumentation process [178]. Adding our sub-categorisations was necessary to enable PseudoSweep to instrument source code in a way that is compilable. For example, in JavaParser, *variable declaration* statements are categorised as *expression* statements. Yet, if we were to delete the statement as we do with other *expression* statements, the instrumentation would introduce compilation errors. Since statements will likely be categorised differently by other tools, future work will investigate how different statement categorisations affect results. Threaded code originating within tests themselves can interfere with our tool's internal thread control, which PseudoSweep uses to time out tests if deletions cause infinite loops. We were forced to omit certain tests (see the replication package), which may have introduced some small inaccuracies into our results. Finally, the results for RQ1 through RQ3 include arid nodes [159]; future work on PseudoSweep will remove them from consideration.

We used PIT, with its commonly adopted "default" mutant set, to generate and evaluate the traditional mutants. However, using a different mutation testing tool or a different version or configuration of PIT may yield different results. Finally, the first author performed the manual analysis following a well-documented process to confirm PseudoSweep's findings and cross-check the results to understand the causes of pseudo-tested statements. Since this manual procedure could have introduced errors, future work will validate these interpretations by discussing them with the developers of each project.

3.5 Results

This section first characterises the project set, then answers each research question.

3.5.1 Project Characterisation

This chapter's study differs from prior work as the project set has varying degrees of coverage. We used PseudoSweep to calculate statement coverage and used JaCoCo's [7] line and bytecode coverage scores to confirm the output. Table 3.1 shows that the statement coverage reported by PseudoSweep ranges from 0.8% to 99%, which is similarly reflected in the JaCoCo line coverage (4% to 99%) and bytecode coverage (4% to 98%).

For the Apache Commons projects used in prior studies, coverage was uniformly high; however, the randomly selected projects exhibited a wide range of coverage scores with

Table 3.2: The numbers of not-covered (N), required (R), pseudo-tested (P), and the total number of elements for methods and statements (N+R+P), and pseudo-tested statements in required methods (PiR) for statements, which is a subset of P. Each horizontal bar shows the proportion of a project's elements in N, R and P (© 2024 IEEE).

	Methods				Statements				
Project	# N	# R	# P	(N+R+P)	# N	# R	# P	# PiR	(N+R+P)
asw4j	87	59	24	170	414	385	17	12	816
authlete-jose	20	31	9	60	273	297	0	0	570
beanunit	3	26	15	44	87	138	50	31	275
chardet4j	22	30	28	80	256	426	0	0	682
chronometrophobia	41	24	0	65	41	44	0	0	85
commons-cli	4	132	4	140	20	705	20	5	745
commons-codec	70	523	32	625	385	3285	263	125	3933
commons-csv	0	144	1	145	12	944	4	2	960
commons-math	834	5243	390	6467	5841	40105	260	119	46206
coodoo-listing	72	9	0	81	640	97	2	2	739
data-conjuror	6	1	5	12	12	14	3	0	29
ext4logback	4	3	0	7	25	20	0	0	45
f3270	202	1	1	204	1289	1	9	0	1299
jargser	2	5	3	10	13	83	0	0	96
json-canonicalization	22	32	11	65	735	367	0	0	1102
jena.serializer	0	2	0	2	4	28	0	0	32
kafka-common-lib	19	6	2	27	68	46	3	2	117
lib-logger	31	2	3	36	77	1	17	4	95
marine	12	0	2	14	43	0	2	0	45
payload-off-loading	9	24	4	37	60	101	27	10	188
pmpersistence	10	17	0	27	72	154	2	2	228
sequencepattern	80	29	5	114	293	200	0	0	493
smartbill-client	14	2	0	16	155	32	13	12	200
solr-cmd-utils	569	202	36	807	5636	1615	29	24	7280
wcomponents-compiler	0	7	0	7	16	46	0	0	62
wildfly-swarm-spi	40	22	2	64	161	106	1	0	268
xml-region-analyzer	0	9	1	10	6	216	0	0	222
Total for Apache Commons Projects	908	6042	427	7377	6258	45039	547	251	51844
Total for Randomly Selected Projects	1265	543	151	1959	10376	4417	175	99	14968
Total for all Projects	2173	6585	578	9336	16634	49456	722	350	66812

f3270 obtaining the lowest coverage scores at 0.8% for both bytecode instruction and line coverage, and *xml-region-analyzer* obtaining the highest coverage of 97% and 97%, respectively. The sizes of the projects evaluated in this study ranged from 29 statements to 46206 statements, and from 7 methods to 6467 methods. Using a varied set of projects enables us to explore pseudo-testedness from different perspectives.

3.5.2 RQ1: How frequent are pseudo-tested elements?

Table 3.2 shows the distribution of "not-covered" (N), "required" (R) and "pseudo-tested" (P) methods and statements for each project. Whilst determining the frequency of pseudo-tested methods enables us to calibrate this chapter's results with prior work, calculating the frequency of pseudo-tested statements characterises a topic that has not previously been studied. For both methods and statements, the table also reports the total number of elements across a project (N+R+P). For statements, we also identified the subset of pseudo-tested statements that our tool found within the required methods (PiR), therefore highlighting a crucial limitation of XMT.

In the following analysis of the data in Table 3.2, we first highlight the overall trends,

before examining the frequency data for the Apache Commons and randomly selected sets of projects.

Frequency of pseudo-tested methods

Results from prior empirical studies suggest that pseudo-tested methods are present in all projects [137, 195, 23]. However, Table 3.2 shows that this trend does not hold for our chosen projects, as seven of the randomly selected projects contained no pseudo-tested methods. With that said, all Apache Commons projects used in prior work contained at least one pseudo-tested method. This result can be explained by the fact that, as mentioned in Section 3.5.1, the chosen project set differs from previous studies, with more varied project types and levels of testedness.

Some projects exhibit very little coverage, some have only required methods, and others have a more even distribution. By splitting covered methods into required and pseudo-tested methods, the results show that 23 of the 27 projects contain more methods that are required than pseudo-tested, with a maximum of 5243 required methods in *commons-math* and a minimum of zero in *marine*. Overall, pseudo-tested methods were present in 20 of 27 projects, with the smaller randomly selected projects tending to have no pseudo-tested methods.

The maximum number of pseudo-tested methods for the Apache Commons projects was 390 in *commons-math*, while in the randomly selected projects the maximum was 36, which PseudoSweep found in *solr-cmd-utils*. The median number of pseudo-tested methods across all projects was three, whereas the median number of required methods was 22; overall, projects tended to have fewer pseudo-tested methods than required methods. All projects contained at least one covered method; therefore, PseudoSweep did not find any projects with both zero required *and* zero pseudo-tested methods.

There were some interesting distributions of method coverage in the randomly chosen projects. For example, *f3270* contained 204 methods of which only two were covered, with one required and one pseudo-tested due to an apparent lack of testing effort. The low coverage in this project showed that even with seven test cases covering the two methods, the test suite required only one of the methods to pass. The *lib-logger* project also had low coverage, with 13.8% ((2+3)/36) coverage split into two required and three pseudo-tested methods, achieved with only three test cases. Other project examples include *wcomponents-compiler* and *jena.serializer*, both of which contained fewer than 10 methods in total, all of which were classified as required. Also, *ext4logback* had fewer than 10 methods but contained four non-covered methods and three required methods. No project with fewer than 10 methods had any pseudo-tested methods.

Frequency of pseudo-tested statements

To our knowledge, the frequency of pseudo-testedness at the statement level has not yet been studied. Prior work evaluates SDL as a mutant reduction technique, with limited study of its use beyond this purpose [49, 162]. Understanding the frequency of pseudo-tested statements across different method types will help us better understand their importance to developers. Table 3.2 shows that the projects contained 66812 statements in total, of which their tests covered only 50178 (49456+722). The set of Apache Commons projects accounts for the majority of statements studied, totalling 51844 statements, whereas the randomly selected projects contributed 14968 statements.

Pseudo-testedness accounted for a significantly smaller percentage of statements than of methods. Pseudo-tested methods comprised 6.1% (578/9336) of the total methods, whereas pseudo-tested statements comprise 1.08% (722/66812) of total statements. For the Apache Commons projects, pseudo-tested statements contributed to 1.05% (547/51844) and for randomly selected projects they made up 1.17% (175/14968). The project set also contained zero pseudo-tested statements for ten projects, revealing that pseudo-testedness at the statement level does not exist in all projects. Yet, across all projects, the median of pseudo-tested statements was two.

PseudoSweep found 350 PiR statements, ranging from zero in 14 projects to 125 in *commons-codec*, a large, well-tested project. The median was also zero, indicating they are uncommon in most projects. However, in the projects where they are present, PiR statements are 48% (350/722) of pseudo-tested statements. These statements are either redundant code or do not influence the outcome of a test suite check, and would also be overlooked by XMT.

Conclusion for RQ1. Pseudo-tested elements (P) make up 6.1% (578/9336) of methods and 1.08% (722/66812) of statements. Pseudo-tested statements within required methods (PiR) are 48% (350/722) of pseudo-tested statements.

3.5.3 RQ2: Do pseudo-tested elements have low mutation scores?

Figure 3.1 provides the traditional mutation scores for all projects that had both required (R) and pseudo-tested (P) methods. We can calculate the scores for individual methods and statements by isolating mutant locations that occur within these regions and calculating the mutation score using only these mutants. Figure 3.2 provides the traditional mutation scores across all projects by PIT mutation operator in required (R) and pseudo-tested (P) methods. Furthermore, Figure 3.3 gives the traditional mutation scores for those projects with both required (R) and pseudo-tested statements in required methods (PiR). Figure 3.4 provides the traditional mutation scores across all projects by PIT mutation operator in required (R) and pseudo-tested statements in required methods

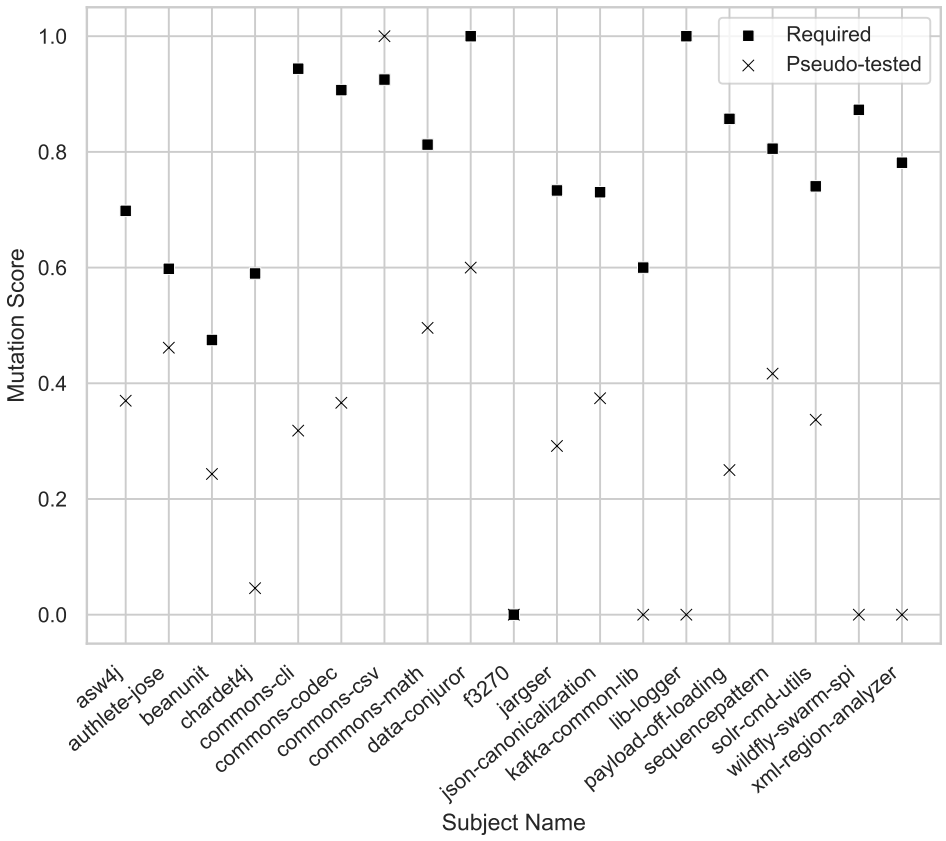

Figure 3.1: Mutation scores for pseudo-tested and required methods (© 2024 IEEE).

(PiR). Finally, for each classification of methods and statements, Table 3.3 provides the number of elements, the total number of mutants, the number of mutants killed by the tests and the overall mutation score, ranging from 0.0 to 1.0 inclusively, as calculated by PIT. In the following analysis of these three data sources, we highlight both the overall and project-specific trends.

Method Mutation Scores

Figure 3.1 shows that the mutation scores for required and pseudo-tested methods follow the trend identified by prior work [195]: those that are pseudo-tested obtain lower mutation scores than those that are required. Table 3.3 reveals that the overall mutation score for required methods is 0.82 but only 0.40 for pseudo-tested methods. Figure 3.1 also shows that, for all projects except *commons-csv*, the mutation scores for the required methods are higher than those of the pseudo-tested methods.

```
1  char[] lookAhead(final int n) throws IOException {
2      final char[] buf = new char[n];
3      return lookAhead(buf);
4  }
```

We attribute the disparate result for the single Apache Commons project to the fact that it had only one pseudo-tested method with a single mutatable statement with a

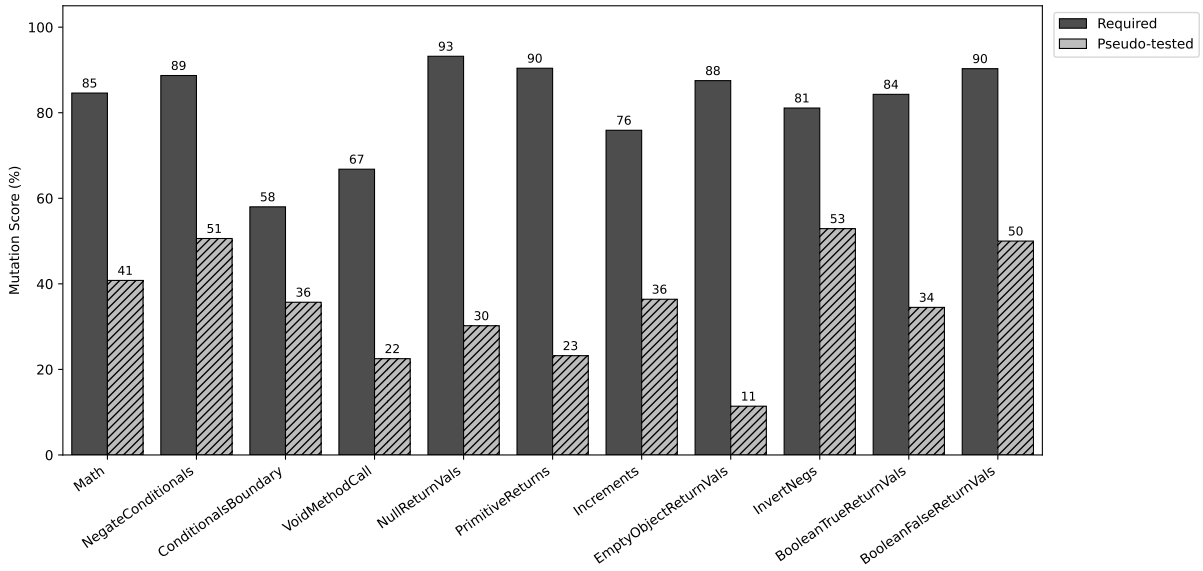

Figure 3.2: Mutation scores for pseudo-tested and required methods by PIT mutation operator (© 2024 IEEE).

single, killed PIT mutant (`return null`), ultimately enabling its test suite to obtain a 100% traditional mutation score. PseudoSweep also returned a *new ArrayList* which the test suite did not detect making it pseudo-tested.

Figure 3.2 shows that for all PIT mutation operators used, the mutation scores for each individual operator was lower in pseudo-tested methods than in required methods. While this is intuitive, it highlights how the coverage alone is not enough to detect these synthetic faults. The most notable differences are in `NullReturnVals` and `EmptyObjectReturnVals` which is expected as they follow a similar path to the method level operators used to identify pseudo-tested methods. However exact operator set for return statements differs with method-level trying two mutation operators for each `return` statement meaning that the scores are not 0% in pseudo-tested methods.

Statement Mutation Scores

PseudoSweep found 14 projects that did not contain any pseudo-tested statements in required methods. Since traditional mutation testing cannot mutate non-existent pseudo-tested statements, Figure 3.3 does not present traditional mutation scores for them. With that said, Table 3.3 shows that the overall mutation score of 0.57 for PiR statements is lower than that of required statements at 0.80, similar to the trend detected at the method level. Figure 3.3 reveals that for two of the four Apache Commons projects (i.e., *commons-codec* and *commons-math*) the mutation score for required statements is higher than for the PiR statements. Note that for *commons-csv* the mutation score for PiR is higher than the mutation score for the required ones since it contains only one mutant in a PiR statement that was killed. For *commons-cli*, we found that the

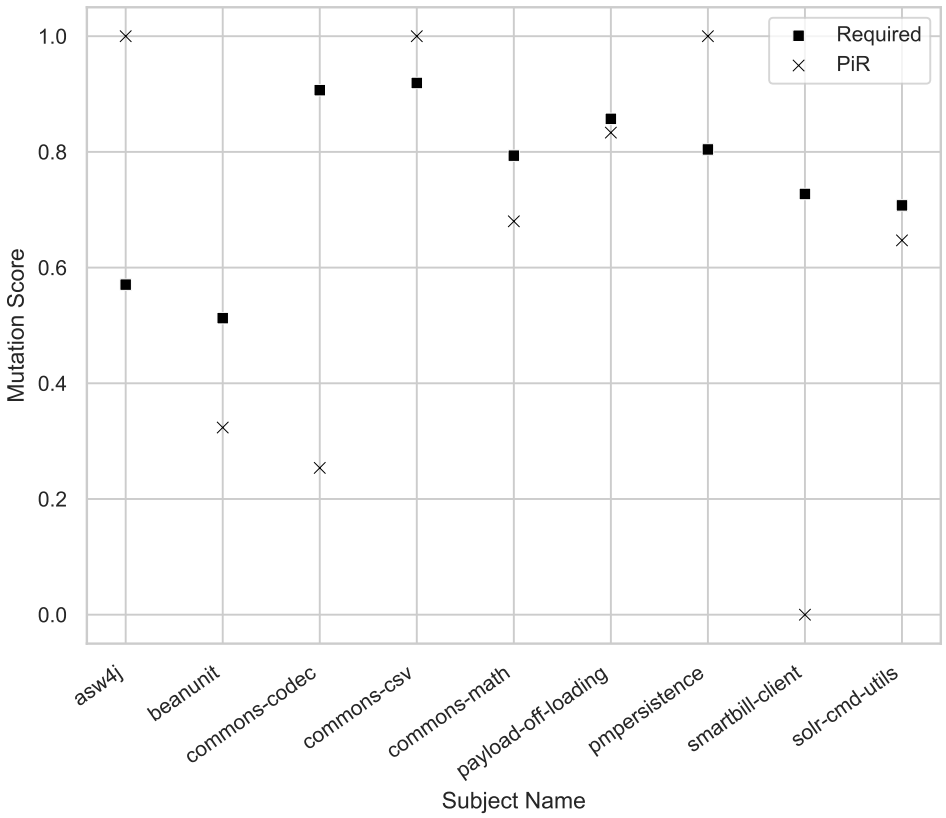

Figure 3.3: Mutation scores for PiR and required statements (© 2024 IEEE).

PiR statements contained no mutants. For the randomly selected projects, the mutation scores for required statements were higher than those for the PiR statements. For *asw4j* and *pmpersistence*, PIT generated only a single, killed mutant in PiR statements, causing the mutation scores for PiR to be 1.0 for both projects and therefore higher than the score for required statements.

Figure 3.4 shows that for all PIT mutation operators used, the mutation scores for each individual operator was lower in PiR statements than in required methods, except for the `Increment` operator. The `Increment Operator` flips increments of "local variables only" for example from `i++` to `i--`. Interestingly, the score for these was 100% with only 4 appearing across PiR statements. Many of the PiR statements contain few mutants, meaning that the addtion of a single killed mutant can largely influence the percentage score.

Conclusion for RQ2. Pseudo-tested methods (P) obtain a significantly lower mutation score of 0.40 than that of the required methods (R) scoring 0.82. Pseudo-tested statements in required methods (PiR) obtain a mutation score of 0.57 compared to the required statements' mutation score of 0.80. When breaking the results down by PIT operator, the trend continues across each operator, highlighting that low mutation scores across pseudo-tested elements was not unique to specific PIT operators.

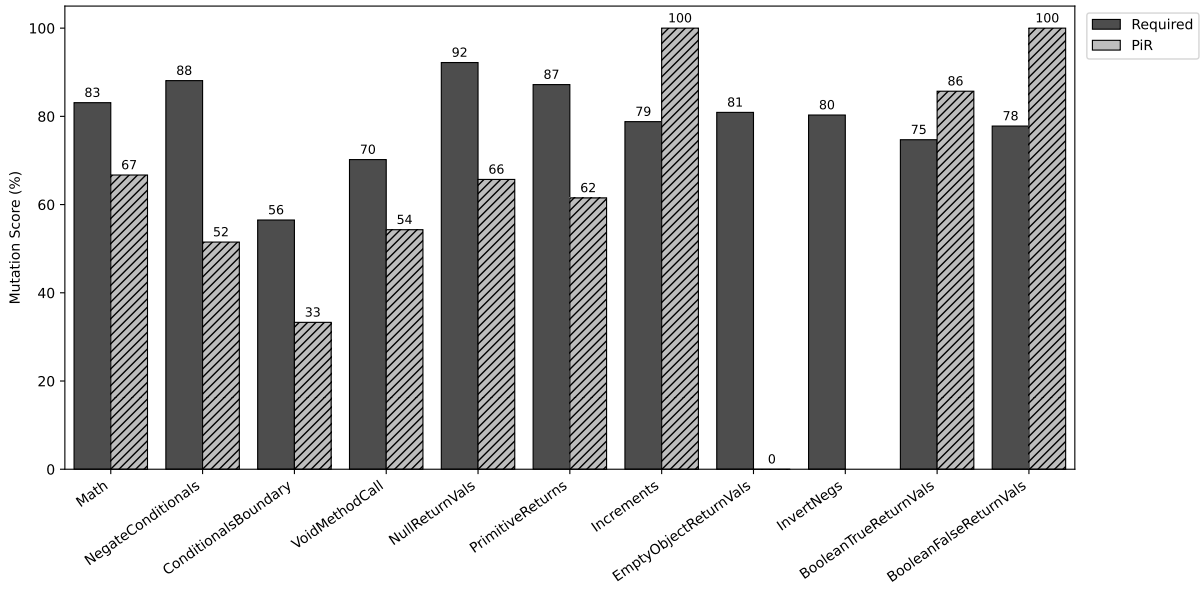

Figure 3.4: Mutation scores for PiR and required statements by PIT mutation operator. Where no score label appears, this is due to 0 mutants being present and therefore no score can be calculated (© 2024 IEEE).

3.5.4 RQ3: Does PIT's default operator set highlight deficient testing with respect to pseudo-tested statements?

Section 3.5.3's answer to RQ2 pointed out that PIT generated very few mutants for the elements that PseudoSweep classified as pseudo-tested. As shown in Table 3.3, this lower "mutants-per-element" count for pseudo-tested elements was evident at both the method and statement levels. At the method level, required methods contained 6.97 (45878/6585) mutants per method, compared to those that were pseudo-tested, which had an average of 3.67 (2124/578) mutants per method. Required statements also had a higher count of 2.31 (114164/49456) mutants per statement as opposed to PiR statements with 0.92 (322/350) mutants per statement. Importantly, if PIT does not generate sufficient mutants for pseudo-tested elements, then traditional mutation testing will not accurately characterise their well-testedness.

For all of the statement types across all of the projects, Table 3.4 shows that the ones with the fewest number of mutants generated for them were: 588 *break* statements containing 4 mutants between them, 153 *continue* statements containing 0 mutants and 1997 *throw* statements containing only 82 mutants between them. This result suggests that there are several types of Java statements for which it is insufficient to assess test adequacy using mutation testing with PIT due to its operation at the bytecode level and limited operator set.

This table also reveals that there are several types of Java statements, such as *do* and *lambda return*, for which there are no pseudo-tested statements in required methods. In this case, PIT could not generate any mutants for these statement types and, therefore,

Table 3.3: For required (R), pseudo-tested (P) and not-covered (N) methods and statements and pseudo-tested in required statements (PiR), the number of program elements (# Elements), number of traditional mutants (# Mutants), number of mutants detected by the test suite (# Killed) and the overall mutation score (Score = # Killed / # Mutants) as calculated by the PIT mutation testing tool (© 2024 IEEE).

Elements	# Elements	# Mutants	# Killed	Score
Methods				
Required	6585	45878	37466	0.82
Pseudo-tested	578	2124	839	0.40
Not covered	2173	7115	219	0.03
Statements				
Required	49456	114164	90973	0.80
Pseudo-tested	722	818	406	0.50
PiR	350	322	184	0.57
Not covered	16634	20071	1971	0.10

the mutation score was undefined (-). Even when there were PiR statements for certain statement types, like *break* and *continue*, PIT did not generate any mutants, and thus the mutation score was also undefined (-). Where PIT did generate mutants for PiR statements, it yielded only 0.92 (322/350) mutants per statement, which is fewer than the 2.31 (114164/49456) mutants across all statements.

This study used the DEFAULT PIT operator set, which was restricted to *Conditionals Boundary*, *Increments*, *Invert Negatives*, *Math*, *Negate Conditionals*, *Return Values*, *Void Method Calls* and *Empty/False/True/Null/Primitive* (returns 0 for numeric value types) return types. While there is some overlap in return type mutation, it is limited as most object returns are replace with `null` which is often killed, which can be seen in the high mutation score in return statements. As shown in Table A.2, PseudoSweep returns targeted types as well as `null` values causing PseudoSweep to return values that do not change the outcome of the test suite. Furthermore, it is notable that *expression* statements contain a proportionately low number of mutatants. This can also be accounted for by PIT's operator set not supplying a range of mutation operators that account for expression statements, as multiple operators target statements found in control flow statements such as *if* and *for* statements. Reproducing this study with a different operator set would, therefore, produce differing results.

Finally, the mutation score for a specific type of PiR statement was always less than the corresponding score for all statements of the same type. For instance, the mutation score for *while* statements was 0.65 across all program statements and only 0.33 for those in PiR, suggesting that pseudo-tested statements in required methods are less well tested than the others.

Table 3.4: Traditional mutation scores for each statement type, for all statements (All) and pseudo-tested statements in required methods (PiR), calculated using the PIT mutation tool [37]. "-" is used where there are no statements to apply mutations or a mutation score cannot be calculated ().

	All				PiR			
Statement Type	# Statements	# Mutants	# Mutants killed	Score	# Statements	# Mutants	# Mutants Killed	Score
break	588	4	1	0.25	9	0	0	-
continue	153	0	0	-	2	0	0	-
do	44	1241	720	0.58	0	-	-	-
expression	22936	17155	11373	0.66	184	98	63	0.64
for	2968	25528	20748	0.81	13	52	33	0.63
for each	731	1750	846	0.48	0	-	-	-
if	16640	58479	37631	0.64	43	91	42	0.46
inner class	39	19	7	0.37	0	-	-	-
inner class return	70	60	52	0.87	0	-	-	-
lambda	31	69	33	0.48	0	-	-	-
lambda return	29	100	61	0.61	0	-	-	-
loop condition	3435	6496	5363	0.83	0	-	-	-
return	9683	12549	9070	0.72	60	53	36	0.68
switch	102	3170	2209	0.70	1	2	0	0
throw	1997	82	6	0.07	16	0	0	-
try	456	1494	539	0.36	0	-	-	-
variable declaration	6466	1743	1383	0.79	17	5	3	0.60
while	444	5114	3308	0.65	5	21	7	0.33

Conclusion for RQ3. When using the default operator set from the GREGOR engine, PIT generated only 0.92 (322/350) mutants per statement for PiR statements, while generating 2.31 (114164/49456) mutants per statement for required statements. Moreover, statement types such as *break*, *continue* and *throw* also yield few PIT-generated mutants. Ultimately, the traditional mutation scores for all statements are higher than those for PiR statements, suggesting that PIT's default mutation operator set does not effectively highlight deficient testing for pseudo-tested program statements.

3.5.5 RQ4: What are the causes of pseudo-tested statements?

We manually studied pseudo-tested statements to identify their root causes and how developers can handle them. In the remainder of this discussion, we present the results of the manual analysis, identifying the specific cause of pseudo-tested statements and the number of times it was evident.

No targeting assertion (70)

A test case covers statements in this category, but no dedicated test assertion is responsible for checking the pseudo-tested statement. This was the most common category we found, containing 70 statements.

One example was a *void* method call in *solr-cmd-utils* to a silent, error-handling method that silently logged exceptions of a specific type. The test suite did not check these logged exceptions; therefore, the *void* method call statement is pseudo-tested. Any calls to pseudo-tested methods will likely be pseudo-tested as well. Thus, this pseudo-tested statement presents little information to the developer.

In *commons-codec*, we identified multiple statements without assertions checking them. An example was the *language.Caverphone2.encode* method, containing 82 lines of code (LOC), which encodes an input string into a "CaverPhone 2.0" value (see Appendix A.3.2). This required method contained 63 statements, of which 23 were pseudo-tested. This encoding process used the *java.lang.String.replace(t, r)* and *java.lang.String.replaceAll(r, r)* method calls to replace specific characters and regular expression patterns within the input string. PIT's default operator set does not mutate these non-*void* method calls, meaning that across all 82 LOC, PIT seeded only three mutants, of which the test suite killed two. This example shows that where a method relies on the returned values of non-*void* method calls, it is easy to miss assertions, as neither code coverage nor PIT's default set would highlight these issues to a developer. The documentation for PIT lists a "Non-Void Method Call" mutator that can be used but is not in the default set. The documentation also notes that this mutator is unstable and may create equivalent mutants. Importantly, statement deletion consistently produces compilable mutants and does not provide equivalent mutants in this context. Developers could address these issues by adding assertions to evaluate the consequences of the pseudo-tested statements.

Partial assertion (7)

We found partial test assertions in which an assertion for the statement was present, but it checked only part of the program's output. For example, PseudoSweep identified pseudo-tested statements related to partially checked string variables. Some statements involved method calls that returned substrings that were combined to form the final output string. When tests only partially check these strings using a "contains" assertion, some statements related to the string are not required for the test suite to pass. PIT's default operator set does not mutate this code and would not necessarily reveal the partial assertion issue. We also found this when the test suite checked only the exception type and no further details, thus showing weak exception handling within the test suites. In *commons-csv*, PseudoSweep identified a *hashcode* method whose tests check that it produces both equal and unequal hashcodes. The pseudo-tested statement in *hashcode* applied the same operation to all input, and, therefore, one could categorise it as a partial assertion or a redundant statement. However, we can not ascertain this without further knowledge of the project.

No targeting test (9)

We found nine pseudo-tested statements caused by the lack of a test aimed to cover the statement. Other tests executed these statements, but the test suite did not include a test specifically designed for the statement behaviour. We identified six pseudo-tested *if* statements, caused by no test executing the *if* condition to "true". This left code within

the *if* statement body uncovered. While this coverage deficiency could be identified by JaCoCo [7], only addressing coverage may not, in turn, address the pseudo-testedness. The other two pseudo-tested statements are unchecked *throw* statements for which no targeting tests existed. We could also have classified these two statements as ambiguous exceptions, but since there was no test aimed at them in the first place, we concluded that this was the primary category for them.

Unintended exception handling (6)

These tests used multiple assertions to check different behaviours within a single test, including an expected exception. If any of these assertions trigger the expected exception (even if unintentionally), the JUnit *expected* exception attribute catches it, and the test continues to pass, thus making the statement pseudo-tested. In *asw4j*, we found two examples of tests that used assertions to consecutively call the same method with different inputs. The tests also contained a JUnit *expected* exception annotation attribute — intended for the last assertion in the test cases, which provided the method with an invalid input. Yet, this annotation caught any exception of the same type thrown by the other assertions. This causes the test to pass when the assertions failed unexpectedly, making the statement pseudo-tested. This instance of a pseudo-tested statement therefore helps developers to identify poorly designed tests such as this.

The categories of *no targeting assertion*, *partial assertion*, *no targeting tests* and *unintended exception handling* identify actionable insights that a developer gains from studying these pseudo-tested statements found by PseudoSweep. For example, adding a test for an unchecked behaviour or adding assertions to check more program state are both intuitive outcomes of identifying pseudo-tested statements. These are also concerns that XMT would not raise, meaning that easy-to-address issues will be left for traditional mutation testing to identify at additional expense or, in the cases where traditional mutation testing does not have an appropriate operator for the statement, the developer may overlook the concern altogether.

Conclusion for RQ4. Pseudo-tested statements within required methods can highlight testing issues that XMT, traditional mutation testing and code coverage do not detect.

3.6 Conclusions

An extreme mutation testing (XMT) tool individually deletes method bodies in covered program code and observes whether the test suite detects their absence [137]. XMT labels as "pseudo-tested" any method that it removes without influencing test outcomes and calls a method "required" if its deletion causes a change in test status. Focusing on

a finer granularity than XMT, this chapter presents the first empirical study of pseudo-testedness in program statements. Using 27 open-source Java projects, the experiments use the PseudoSweep tool to perform statement deletion (**R1**), revealed:

1. XMT would overlook 48% of the pseudo-tested statements since they appear in required methods (**A1**).
2. Pseudo-tested statements were also locations of lower mutation scores, suggesting that such statements are points of test suite weakness (**A2**).
3. PIT's default set of mutation operators generated significantly fewer mutants for certain statement types, meaning that neither traditional nor extreme mutation testing will likely highlight issues within these statements (**A2**).
4. A further manual analysis of 119 pseudo-tested statements across 30 methods identified testing concerns that code coverage, extreme mutation testing and traditional mutation testing did not fully identify (**A3**).

While this information is useful to developers for identifying missing test assertions, its effectiveness among other oracle-based test adequacy criteria remains unclear. As such, this necessitates a comparative evaluation, as presented in Chapter 4.

Chapter 4

Empirically Analysing Oracle Gaps in Covered Code

The contents of this chapter are based on the paper "M. Maton, G. M. Kapfhammer, and P. McMinn. Where Tests Fall Short: Empirically Analyzing Oracle Gaps in Covered Code. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2025" ().

Author Contribution to Publication For this publication, I was responsible for implementing Checked Coverage with a dynamic slicer and an observational slicer, along with the experimental design, data collection and preparing the manual analysis, which was then undertaken by all authors collectively. I drafted the paper under the guidance and supervision of the second and third authors. The second and third authors made substantial textual edits to the final version.

Use in this Chapter For this chapter, the content of the original paper has been adapted and edited to align with the background provided in Chapter 2 and to define the terminology for this chapter. To maintain a consistent voice throughout this thesis, I have independently re-authored all sections drafted by co-authors.

4.1 Introduction

Chapter 3 revealed identifying pseudo-tested statements to be an effective technique for locating areas of low mutation score. Similarly, recent evidence has found that identifying *oracle gaps*, that is, test-executed code (i.e., covered) that, although executed, does not cause a test case to pass or fail, may be a cost-effective way to find where tests fall short [80]. Intuitively, the test suite does not adequately test the code within these gaps [80]. Existing oracle gap calculation approaches (OGCAs), such as host checked coverage, extreme mutation testing (XMT) and identifying pseudo-tested statements (see

Chapter 3), highlight covered code that is unchecked by an oracle (e.g., a test assertion) to provide developers with clear testing direction without the expense of equivalent mutant analysis and execution times [80, 129, 137].

Whilst each OGCA has been given varying attention, there exists no empirical evidence to guide developers in choosing the appropriate one [80, 129, 137, 195, 196, 23]. As such, developers remain uninformed regarding suitable oracle gap sizes; the calculation costs of these gaps; the statement types within the gaps; and the mutation scores of the statements in the oracle gaps. For developers, answering these questions is critical. An empty gap offers no testing direction, while an overly large oracle gap is not cost-effective for developers to address. Furthermore, if an OGCA places irrelevant statements in the oracle gap, then the developer may waste time strengthening tests for unimportant code. Conversely, low mutation scores across oracle gap statements highlight a test suite with low fault detection, necessitating targeted test improvements.

This chapter performs a knowledge-seeking study to analyse and compare the statements within the oracle gaps identified by three OGCAs (**R2**): checked coverage using a dynamic slicer (CC_{DS}); checked coverage using an observational slicer (CC_{OS}) and pseudo-tested statement identification (PTSI) for 30 Java classes across six open-source projects, including libraries and SDKs, ranging in program size and test coverage. To support this study, we present a generalised oracle gap definition that enables comparison between them. This chapter's empirical results will help developers pick the right OGCA to make the most effective use of their limited testing time.

We quantitatively evaluate the different oracle gaps in terms of prominence, statement distribution, correlation with fault detection and execution time. A manual inspection with negotiated agreement revealed that data loading statements, iteration statements and output updates were the most prominent statement types and patterns (**B3**). Generally, the differences between the statements in the oracle gaps indicate that the OGCAs cannot be used interchangeably. PTSI highlighted code locations with the lowest median mutation scores of 0.32, compared to CC_{DS} (0.76) and CC_{OS} (0.50), therefore revealing priority testing areas for improving fault detection. PTSI also had the shortest median execution time of 19.9 seconds, compared to 273.2 seconds (CC_{DS}) and 5957.1 seconds (CC_{OS}), suggesting it could be a good initial test suite analysis approach for developers to quickly identify weak testing areas (**B2**). The contributions of this chapter are, therefore, as follows:

B1 A generalised definition of the oracle gap. Existing techniques use a range of terms, such as "coverage gap" and "pseudo-tested statements". We present a unified definition to enable comparative analysis of techniques.

B2 A quantitative evaluation of the oracle gap prominence, distribution, fault detection rate and execution times using checked coverage with dynamic and observational

Motivating Example
□ Inconsequent to assertion pass/fail ■ Contributes to assertion pass/fail

DOR — *Oracle-based adequacy criteria*

```
1 □■□ public String createXmlTag(String name,
2 □■□                             String content) {
3 ■□□   if (name == null || name.trim().isEmpty()) {
4 □□□       System.err.println("/* ... */");
5 □□□       return "";
6 □□□   }
7 ■■□   String s = "<" + name + ">";
8 ■□□   s += content;
9 ■■■   s += "</" + name + ">";
10 ■■■  return s;
11 □■□ }
12
13   @Test
14   public void test_checkEndTag() {
15       String tagName = "message";
16       String tagContent = "Hello, world!";
17       String result = createXmlTag(tagName,
     tagContent);
18       assertTrue(result.contains("</message>"));
19   }
```

Listing 4.1: Motivating example using prepared Java method `createXmlTag` and its accompanying JUnit test. Column headers: **D**: Dynamic Slice; **O**: Observational Slice; **R**: Required (not pseudo-tested). The highlighted text is "covered" under statement coverage. Statements that are covered but do not contribute to an assertion passing or failing are in the oracle gap (© 2025 IEEE).

slicing and pseudo-tested statement identification.

B3 A qualitative manual inspection of code patterns identified as ineffectual by each technique.

A complete replication package for the paper this chapter is based on is available at [5], containing all tools, execution data, manual inspection decisions and project links to enable replication and extension of this work.

4.2 Defining the generalised Oracle Gap

The motivating example in Listing 4.1 illustrates how using different techniques to evaluate an oracle, in this case, the assertion on Line 18, can yield conflicting oracle gaps. The example test suite contains a single test case for the Java method `createXmlTag`, with the highlighted grey lines representing the statements the test suite covers. The columns of boxes represent the statements on a given line that determine whether the assertion on Line 18 passes or fails, as identified by the three techniques. For example, column **D** represents a dynamic slice from the assertion, for which the slice contains the statements on Lines 3, 7, 8, 9 and 10 (marked by ■). Conversely, column **O**, which shows an observational slice, contains Lines 1, 2, 7, 9, 10 and 11. Interestingly, the statements on Lines 3 and 8 are not on the observational slice (marked by □) despite being covered (i.e., highlighted grey). As such, these statements fall within the oracle gap between statement coverage and the observational slice, as although covered, the observational

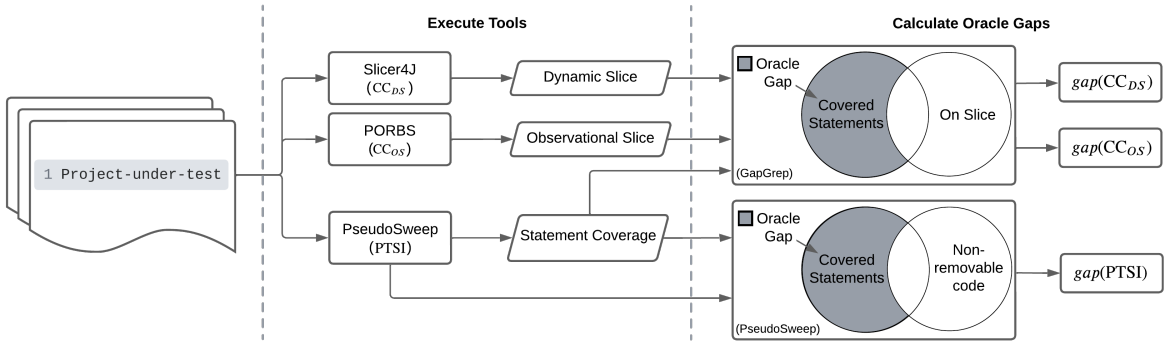

Figure 4.1: Identifying Oracle Gaps with GapGrep: Oracle gaps are calculated by combining the outputs of Slicer4J (CC_{DS}) and PORBs (CC_{OS}) and statement coverage. PseudoSweep (PTSI) calculates its oracle gap (i.e., the pseudo-tested statements) internally; however, we visualise it separately for clarity (© 2025 IEEE).

slice does not include them as contributing to the assertion passing or failing. To remove the statements from the gap, a developer should write additional tests that evaluate the tag content and handling of invalid inputs. The dynamic slice, however, contains all of the covered statements and, therefore, has an empty oracle gap, implying no further tests are needed. Column **R** shows required statements (i.e., not pseudo-tested), which are covered statements that, when individually deleted or set to a different value, will cause a test to fail (as marked by ▪). In this example, required statements create the largest oracle gap (i.e, statements on lines 3, 7 and 8). We can see that required statements do not include the string initialisation on line 7 as it does not matter what value is assigned during initialisation for the test case to pass. The dynamic and observational slice, on the other hand, consider whether the statement as a whole is necessary for the test case to pass (i.e., it is necessary to declare and initialise the string) and, therefore, it appears on both of these slices.. The discrepancy between the oracle gaps leaves developers in the dark, unsure which code is tested and which gaps are actually useful for improving their testing.

4.2.1 Oracle Gaps

As described in Section 2.4, test oracles, such as assertions, are invaluable for assessing the expected program behaviour. Recent studies have taken a promising approach to identifying missing assertions for covered code, however these approaches use differing terminology and definitions. Each of these Oracle Gap Calculation Approaches (OGCAs) identifies covered code that does not fulfil a secondary oracle assessment criterion, as shown in Listing 4.1. Hossain et al. defined the coverage gap as the set of elements executed by tests but not observed by a test oracle [80]. In extreme mutation testing (XMT), pseudo-tested methods are executed by the test suite, yet the method body is removable without causing test failures [137, 195]. Chapter 3 extended this to the

statement level, finding pseudo-tested statements in non-pseudo-tested methods [129]. For this study, we generalise this as an oracle gap as defined below (**B1**).

Definition 4.2.1 (Generalised Oracle Gap)**,** All covered statements that do not cause a test oracle to pass or fail.

This definition divides covered code into two categories:

Definition 4.2.2 (Effectual Statement)**,** A covered statement that causes a test oracle to pass or fail.

Definition 4.2.3 (Ineffectual Statement)**,** A covered statement that does **not** cause a test oracle to pass or fail.

This section explains how checked coverage using dynamic slicing (CC_{DS}), checked coverage using observational slicing (CC_{OS}) and pseudo-tested statement identification (PTSI) fit under these classifications. We refer to their respective oracle gaps using *gap*(CC_{DS}), *gap*(CC_{OS}) and *gap*(PTSI).

4.2.2 Checked Coverage using Dynamic Slicing (CC_{DS})

The OGCA using Checked Coverage using Dynamic Slicing (see Section 2.4.3 for full definition) is labelled CC_{DS}. The *gap*(CC_{DS}) is the set of covered statements that do not appear on a dynamic slice, and, as such, are ineffectually covered. In Listing 4.1, *gap*(CC_{DS}) is empty, as all covered statements also appear on the dynamic slice.

4.2.3 Checked Coverage using Observational Slicing (CC_{OS})

Like dynamic slicing, observational slicing (as defined in Section 2.4.3) creates a slice using combined deletions that can be used to calculate checked coverage and determine whether program code influences test oracle outcomes.

Definition 4.2.4 (Checked Coverage using Observational Slicing)**,** Checked Coverage using Observational Slicing requires each program line $l \in P$ to be on at least one observational backward slice from an explicit test suite check.

It is worth noting that an observational slice is calculated in terms of lines; however, the oracle gap refers to the statements on those lines. In terms of the *gap*(CC_{OS}), the effectual statements are those covered and appearing on lines on a slice (i.e., not deleted), whereas the ineffectual statements are those that are covered yet do not appear on a slice (i.e., deleted). The *gap*(CC_{OS}) comprises the originally covered, but collectively deletable lines (i.e, the set of statements on ineffectual lines).

4.2.4 Pseudo-tested statement identification (PTSI)

The set of pseudo-tested statements (as defined in Section 3.2) forms *gap*(PTSI). PTSI individually removes each statement, or where necessary for compilation, supplies default values as substitutes and then evaluates whether the test suite notices the missing statement or statement value. The statement is then added back, and PTSI evaluates the next statement. However, when using CC_{OS}, if the line deletion is successful (i.e., no tests fail), it keeps the deletion and tries the next line. As such, CC_{OS} deletes lines collectively to find the minimal set of lines necessary for the test suite to pass.

4.3 Evaluation

This section states and contextualises the five research questions this chapter addresses regarding the calculation of the oracle gap. As there is no existing comparable evidence, we must first quantify the oracle gaps calculated by each OGCA, enabling us to compare and contrast them in subsequent research questions. This includes the size of the gaps, and the similarity between them. A low similarity in gap content suggests the techniques are not interchangeable (i.e., each has a unique use case). This directly leads to research questions 1 and 2:

RQ1: Prominence – What proportion of a subject's production code lies within the oracle gap calculated by each approach?

RQ2: Distribution – How do the contents (i.e., program statements) of each approach's oracle gap differ and/or overlap?

After understanding the breadth of the oracle gaps, we can examine the individual statements declared ineffectual by each OGCA. Analysing the code statements and patterns identified by each OGCA is critical to evaluating their potential use cases. Furthermore, if developers are to use oracle gaps as an intermediate step towards, or instead of, mutation testing, the gaps must effectively highlight priority areas that need further testing. To reflect this, we ask research questions 3 and 4:

RQ3: Categorisation – Which program statement types appear within the oracle gaps calculated by each approach?

RQ4: Fault Detection – How well does each oracle gap calculation approach highlight areas of low mutation score?

Finally, as developers typically focus on a single class when writing unit tests, it is essential to be pragmatic regarding each approach's performance. As such, we ask research question 5:

RQ5: Performance – How do the execution times of different oracle gap calculation approaches compare?

Answering these questions will enhance understanding of the oracle gaps and how different approaches may be used.

4.3.1 Tooling

We use statement coverage and an oracle-based adequacy criterion to form an OGCA to identify an oracle gap. Statement coverage is of the most similar granuality with oracle-based adequacy criteria we evaluate and, as such, it is a reasonable metric to compare them. Using branch coverage, for instance, would have fewer overlapping elements with each criterion and, therefore, would not be a valuable comparison. We limit the study to statements within methods to ensure comparability across tools, and, therefore, use PseudoSweep's statement coverage as the coverage tool to calculate oracle gaps [130]. To implement CC_{DS}, we used the Slicer4J dynamic slicer for Java, replicating other studies of checked coverage, to enable the analysis of projects using Java 8 or 9, and overcome some limitations of JavaSlicer [11, 9, 110]. For PTSI, we used PseudoSweep, as described in Appendix A, to identify stand-alone pseudo-tested statements within the study projects [130]. Finally, to evaluate CC_{OS}, we use the PORBS (Parallelised ORBS) Observational Slicer implementation [87]. We use PIT, a state-of-the-art Java mutation testing tool [37] to evaluate whether the oracle gaps reveal areas of low fault detection. We implemented a tool called GapGrep, as overviewed in Figure 4.1, to run each OGCA and report and analyse its output in a unified manner. We use GapGrep, available in the replication package [5], to calculate and analyse the oracle gaps and report the results.

4.3.2 Projects

Our targeted projects consisted of open-source Java projects compatible with our chosen OGCA tools. As such, each project must use Java 8 or 9 (Slicer4J and PseudoSweep), JUnit 4 or 5 and be a single-module Maven project (PseudoSweep). Projects should also avoid threading, which can lead to inaccurate results in PseudoSweep [130]. All selected projects must also compile and have a passing test suite.

To ensure this study evaluated a range of projects, we systematically selected projects from the Maven Central Repository, using the OSSF Criticality Score to gauge the most influential and important projects [148, 2]. To promote diversity in the project set, we selected the top five Java projects for each Criticality Score parameter (both positively and negatively weighted), and the highest GitHub Stars count. The parameters of the criticality score, using the original "Pike" configuration, include GitHub repository statistics, such as how recently it was maintained and the number of its dependents. This initial

process yielded 70 projects from the 14 categories, each containing five projects. However, initial experiments revealed that OGCA execution time was a significant constraint, with some individual Java class experiment runs taking multiple days. To ensure a feasible project set, we randomly selected six projects from our initial list. From each project, we identified five class and unit test class pairs, resulting in 30 classes under test. We include a complete list of the classes studied in Table 4.1. Of the projects selected, facebook-java-business-sdk obtained the highest criticality score, inutils4j had the highest contributor involvement, euclid was the least updated, eo-yaml had the highest user activity in the last 90 days and finally java-string-similarity and tabula-java had the highest star counts.

4.3.3 Method

We used GapGrep in these steps to answer the five research questions.

Execute Tools

As shown in Figure 4.1, to calculate the oracle gaps for each of the three OGCAs, GapGrep takes PseudoSweep's statement coverage calculation (i.e., statement coverage in method bodies only) and identifies which covered statements do not fulfil each approach's oracle-assessment criterion.

To ensure a feasible analysis, we focused solely on unit test classes directly targeting the corresponding class-under-test. For instance, our analysis of the Skip class only considered tests within the SkipTest class. Consequently, the coverage data for each class, as presented in Table 4.1, only reflects statement coverage achieved by its dedicated test class.

With GapGrep, we run Slicer4J (CC_{DS}), PORBS (CC_{OS}), PseudoSweep (PTSI) and PIT (mutation testing) on each class and test-class pair. For CC_{DS}, we collate the dynamic slices obtained from each JUnit 4 or 5 assertion into a single set. For CC_{OS}, we collect the list of deleted lines from the class under test, and for PTSI, we collect the statement location, statement coverage and the pseudo-tested statements list. Finally, after executing the test suite for each class, we record the mutations introduced to the program during mutation testing with PIT, along with their locations and status (i.e., uncovered, killed, or survived).

Calculate Oracle Gaps

Using GapGrep, we collate the slices and PseudoSweep's data to identify the ineffectually tested statements that make up the oracle gaps, as shown in Figure 4.1. For CC_{DS} and CC_{OS}, we take the respective program slices and, using PseudoSweep's statement coverage calculator, identify the covered statements that are not on the slice. This forms

each OGCA's oracle gap. PseudoSweep uses internal coverage calculation and thus does not need additional tooling to calculate the PTSI oracle gap (see Appendix A).

Quantify Overlaps and Differences in Oracle Gaps

We compare the sets of program statements in each of the three oracle gaps using Jaccard Similarity, a metric that computes similarity between sets that may differ in size. Jaccard Similarity is calculated by counting the number of source code statements that appear in both sets and dividing by the total number of items in either set. Therefore, the Jaccard Similarity J for two sets of program statements, A and B, is calculated by

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.1}$$

We use the Jaccard Similarity score to compare pairs of sets of varying sizes and quantify the overlap between the different oracle gaps.

Characterise Gap by Statement Types

To contextualise the contents of each oracle gap, we performed a manual study of the statements within the union of the oracle gaps for six classes (one from each project under test). For this, we used a negotiated agreement approach as used in other software engineering studies [31, 78, 155]. Following this method, each author individually assessed the purpose of each statement within each oracle gap before coming together to discuss the assessments and reach a unanimous decision on each statement's purpose. By having multiple authors reach individual judgements, we could ensure authors used their initial impressions based on personal expertise without being influenced by other authors. The discussion that followed then took each of these judgments into account to form a collective decision, benefitting from combined expertise of the group. We then used the agreed-upon assessments to characterise the code patterns in each gap and highlight how each OCGA differs.

Calculate Mutation Scores for Each Oracle Gap

Using the PIT mutation tool for Java, with its STRONGER group of 13 mutation operators, we calculate the mutation score for each class, given its unit tests [10]. We then identify all the mutants, the line on which they occurred and whether they were killed, surviving, or uncovered. Using this, we could then identify the mutants placed in oracle gap statements and calculate the mutation score for each oracle gap for each class.

Compare Oracle Gap Calculation Performance

We calculate the execution time for each tool (including PIT) by timing how long it takes to execute on the class-under-test. For PseudoSweep, this is an overestimation as the run script includes the calculation of pseudo-tested statements, whereas for Slicer4J and PORBS, GapGrep calculates this afterwards.

4.3.4 Threats to Validity

External

This study leverages 30 open-source Java classes that fulfilled a range of criteria. Balancing feasibility for research purposes and practical use cases is challenging, and as a result, computational expense necessarily limited the number of projects and classes. Although this set of classes limited the study's breadth, it facilitated a deeper evaluation of the oracle gaps. Since our study was also restricted to Java projects, its results may not be generalised to other project sets using different languages, library versions, or threaded projects. Furthermore, our evaluation considered three approaches to calculating the oracle gap. Other strategies and implementations may yield varying results, and further studies should explore this. This study also only considered statement coverage as the base coverage metric. Due to the nuances of each technique, it is possible that the trends found in this study may change given a different coverage metric, such as line coverage. For mutation testing, we used the PIT mutation testing tool for Java and its STRONGER mutant set to generate and evaluate traditional mutants [10]. Yet, a different mutation testing tool or mutant set would have yielded different results.

Internal

We use Slicer4J, PORBS and PseudoSweep to implement each OGCA. However, these implementations may not be error-free. Despite our best efforts, our implementation of checked coverage using Slicer4J and PORBS may contain defects that could impact the results. Both Slicer4J (CC_{DS}) and PseudoSweep (PTSI) manipulate the project-under-test by instrumenting source code (PseudoSweep) or moving test classes (Slicer4J), which could lead to an inaccurate oracle gap calculation if the original project behaviour is not retained. Mistakes are also possible in GapGrep's amalgamation of tool outputs for evaluation. We therefore include the tool code and additional scripts in a replication package for transparency, enabling researchers to replicate and extend this work [5].

Construct

The assumption that the mutation score is an indicator of test quality underpins RQ4 and the conclusions drawn from the results. This assumption is a notable threat to

Table 4.1: The counts of elements fulfilling each criteria (© 2025 IEEE).

	Statement Coverage			CC_{DS}			CC_{OS}			PTSI	
project / Class	#Stmts	#Cov	%Cov	On Slice	#Eff	#Ineff	On Slice	#Eff	#Ineff	#Eff	#Ineff
euclid / Real3Range	36	33	92	39	28	5	104	32	1	32	1
euclid / Int2Range	54	30	56	26	22	8	78	25	5	30	0
euclid / IntSquareMatrix	72	65	90	48	40	25	146	62	3	62	3
euclid / Line2	151	118	78	107	98	20	260	106	12	116	2
euclid / Transform2	213	90	42	50	38	52	173	58	32	86	4
inutils4j / MyNumberUtils	5	4	80	4	4	0	17	4	0	3	1
inutils4j / MyMapUtils	4	4	100	4	4	0	50	4	0	4	0
inutils4j / MyReflectionUtils	29	18	62	0	0	18	63	14	4	18	0
inutils4j / MyArrUtils	65	20	31	9	9	11	92	20	0	19	1
inutils4j / MyStringUtils	1025	476	46	388	384	92	1041	131	345	436	40
eo-yaml / JsonYamlDump	10	10	100	13	10	0	34	10	0	10	0
eo-yaml / ReadYamlStream	14	14	100	17	11	3	65	13	1	14	0
eo-yaml / WellIndented	26	21	81	3	2	19	57	19	2	21	0
eo-yaml / Skip	17	17	100	4	3	14	90	15	2	17	0
eo-yaml / ReadYamlMapping	68	56	82	74	51	5	265	53	3	56	0
facebook-java-business-sdk / HashedListAdaptor	34	26	76	1	0	26	70	24	2	26	0
facebook-java-business-sdk / BatchProcessor	12	12	100	3	3	9	116	10	2	12	0
facebook-java-business-sdk / CAPIGatewayEndpoint	38	17	45	17	15	2	92	12	5	17	0
facebook-java-business-sdk / ServerSideApiUtil	103	65	63	38	38	27	176	48	17	64	1
facebook-java-business-sdk / Event	313	85	27	153	85	0	1694	33	52	85	0
tabula-java / Cell	20	6	30	7	6	0	37	4	2	4	2
tabula-java / Line	28	19	68	12	10	9	61	18	1	19	0
tabula-java / Table	27	15	56	17	11	4	74	9	6	8	7
tabula-java / Rectangle	55	39	71	36	29	10	141	38	1	34	5
tabula-java / TextElement	116	80	69	37	34	46	173	56	24	80	0
java-string-similarity / LongestCommonSubsequence	22	20	91	10	9	11	45	12	8	20	0
java-string-similarity / SorensenDice	17	17	100	9	9	8	49	12	5	17	0
java-string-similarity / OptimalStringAlignment	26	26	100	14	11	15	50	16	10	26	0
java-string-similarity / RatcliffObershelp	34	34	100	21	21	13	68	30	4	34	0
java-string-similarity / NGram	48	43	90	40	36	7	52	9	34	43	0
Total	2682	1480	N/A	1201	1021	459	5433	897	583	1413	67

the construct validity of this study since there is conflicting evidence on this relationship. However, our choice is supported by studies demonstrating a correlation between the mutation score and fault-detection effectiveness [16, 98, 153]. Individual researchers' expertise may have biased the discussion within our manual study using negotiated agreement. We mitigated this by having each author reach individual conclusions before the discussion, where each author could justify their reasoning before coming to a collective decision.

4.4 Results

4.4.1 RQ1: Prominence

Table 4.1 shows the distribution of covered, on-slice (CC_{DS} and CC_{OS}) and required lines (PTSI) within the project classes. For each technique, we identify the oracle gaps in each class. We first present the statement coverage data calculated using PseudoSweep for the class, given the paired unit test class. We use this to identify the effectual statements (i.e., those that are covered *and* appear on the slice), and the ineffectual statements (i.e., those that are covered but *do not* appear on the slice). The sum of effectual and ineffectual statements for each technique will always equal the total covered statements. The slices are calculated using lines rather than statements, but are paired with the appropriate

statement to calculate oracle gaps. PTSI calculates required (i.e., effectual) statements differently, as a required statement must be covered by definition. Therefore, there is no separate slice counterpart for this technique. The following analysis of Table 4.1 presents the overall trends, followed by a deeper exploration of the prominence of oracle gaps highlighted in individual classes.

Prominence of Dynamic Slice Oracle Gap ($gap(\mathbf{CC}_{DS})$)

The $gap(\mathrm{CC}_{DS})$ contains statements covered under statement coverage that do not appear on a dynamic slice from a test assertion. Overall, 1201 statements appeared on at least one of Slicer4J's dynamic slices from the test class that corresponded with the class-under-test. Of the covered statements, 1021 also appeared on the dynamic slice and were therefore effectually covered. This left 459 (31%) statements that the test suite ineffectually covered and therefore placed in $gap(\mathrm{CC}_{DS})$.

Looking at the CC_{DS} data in Table 4.1, we can see that On Slice does not necessarily mean covered. The statement coverage used only evaluates statements within methods; therefore, it will not account for elements on the slice beyond this. We can see that the number of effectual elements is greater for most classes than the number of ineffectual elements. For the 10 classes where the ineffectual count is greater than the effectual count, there does not appear to be a correlation with the relative slice size or coverage. Notably, at least one such class appeared in each project under test. We also observe that MyReflectionUtils, HashedListAdaptor and Event are the only classes that contain 0 effectual statements under CC_{DS}. Furthermore, despite 8 classes achieving 100% coverage, only 2 of them have no oracle gap (i.e., 0 ineffectual statements).

Prominence of Observational Slice Oracle Gap ($gap(\mathbf{CC}_{OS})$)

The observational slices are generated by observing the test classes' pass/fail behaviour using PORBS. Suppose a test is unsuccessful (i.e., does not compile, fails, or causes the throw of an unhandled exception) after deleting a line. In that case, the line is an observed dependency contributing to the test case passing. Overall, the $gap(\mathrm{CC}_{OS})$ contained the most statements, at 583 ineffectual statements, accounting for 39% of covered statements. However, this trend does not represent the project set, as the 345 ineffectual statements in MyStringUtils bias this overall value — excluding this class lowers the overall CC_{OS} ineffectual rate to 23.7%. For all but five Java classes, the $gap(\mathrm{CC}_{OS})$ was smaller than or equal to that of $gap(\mathrm{CC}_{DS})$. Interestingly, the observational slice accounts for significantly more lines than are covered (5433 total slice lines compared to 1480 covered statements), highlighting an inefficiency when calculating oracle gaps. For example, in Real3Range, 33 statements are covered, yet 104 lines are on the slice, and in only 1 ineffectual statement. The Event class also had a large slice to coverage ration with 1694 lines of the slice, but

only 85 covered statements, demonstrating a high noise to signal ratio. We can attribute these to how observational slicing constructs its slice, using observed dependencies, including any Java lines required for compilation and contributing to successful runs. At a project level, we can see the differences in oracle gaps as calculated by CC_{OS}. The eo-yaml project had only 8 ineffectual statements across each of the five classes, whereas java-string-similarity despite a similar number of covered statements (118 and 140 respectively). In this sense, the observational slice is a more complete program slice; however, for this use case, not all of the slice provides relevant information when calculating a slice for an OGCA. In practice, this means that $gap(CC_{OS})$ provides a lot of noise while also evaluating irrelevant statements, making it difficult for developers to evaluate what it useful.

Prominence of Required Statements Oracle Gap (*gap*(PTSI))

The key difference between PTSI's required (i.e., effectual) statements and slices lies in how they contribute to test outcomes. Slices are statements that collectively cause a test to pass or fail, whereas, for required statements, PTSI evaluates their individual impact on the test outcome. Across all statements, only 67 were pseudo-tested (i.e., ineffectual), far fewer than for the slicing-based techniques and just 5% of the covered statements. For all classes, there were more effectual statements than ineffectual, with only 11 classes having any ineffectual statements. MyStringUtils had 40 ineffectual statements, but its source code size is larger than the other classes.

Conclusion for RQ1. All OGCAs identified oracle gaps, but their sizes varied from 67–583 statements. The $gap(CC_{OS})$ often contained more sliced statements than covered statements, exposing computational inefficiency. The *gap*(PTSI) revealed a remarkably smaller gap of 67 statements, suggesting it is a less sensitive oracle-adequacy criterion.

4.4.2 RQ2: Distribution

To evaluate the distribution of the statements within the oracle gap, we calculate and present the Jaccard Similarity scores for each pair of oracle gaps. A Jaccard Similarity score of 1 indicates the elements of two sets are identical, whereas a score of 0 would indicate no common elements. In terms of oracle gap content between the different pairs of techniques, there are varying degrees of overlap. The $gap(CC_{OS})$ and $gap(CC_{DS})$ pair exhibit a similarity score of 0.21, which is a higher similarity than for any of the other pairs. Furthermore, the $gap(CC_{OS})$ and *gap*(PTSI) similarity is low at just 0.09, with $gap(CC_{DS})$ and *gap*(PTSI) being even lower at 0.06. The low similarity between *gap*(PTSI) and those of the other OGCAs is likely a reflection of the difference in statement set sizes. For instance, as noted in RQ1's response, *gap*(PTSI) contained 67 statements, compared

to 583 in $gap(\text{CC}_{OS})$. Due to the difference in gap size between PTSI and CC_{OS}, the theoretical maximum Jaccard score is capped at 0.115. Therefore, the actual score of 0.09 indicates that around 78% of ineffectual PTSI statements are also in the CC_{OS} gap. As both CC_{OS} and PTSI are deletion-based techniques, it is intuitive that their gaps contain overlapping statements. The important thing to note is that no one OGCA identifies a super set of the other techniques oracle gaps, demonstrating that these techniques cannot be used interchangably to identify a single oracle gap.

Conclusion for RQ2. The statement sets in the oracle gaps identified by each tool are dissimilar. The $gap(\text{CC}_{DS})$ and $gap(\text{CC}_{OS})$ pair are most similar (0.21), while CC_{DS} and PTSI are least similar (0.06). This is likely a reflection of the difference in set sizes between the oracle gaps. Overall, this highlights that no one OGCA can be used instead of the others and, therefore, it is important to understand which types of statements appear in each oracle gap.

4.4.3 RQ3: Categorisation

Intuitively, the statements in the oracle gaps represent the program source code that, according to each OGCA, was not well tested. To better characterise the statements for which the tests fall short, the three authors performed a manual study of the purposes of the code in the oracle gaps, using negotiated agreement to focus on 454 statements across 6 classes. We present our categorisation of the statements in Table 4.2, along with examples to describe the code patterns in the oracle gaps. When possible, we preserve the original code and indentation, adjusting and contracting it as needed for readability. As described in Section 4.2.1, ineffectual statements are statements that are covered, but do not cause a test oracle to pass or fail as defined by the specified OGCA.

Checks (26)

We defined check statements as code that uses a conditional to evaluate a condition and change the program's execution path. Such ineffectual checks were identified across the oracle gap from each OGCA.

Special Cases were the most common checks to appear in the oracle gaps. These include code structures such as edge-case checks, complex-conditional checks and equality checks.

State Checks included checking flags and markers to determine the program's path. WellIndented, for example, used these checks on ineffectual Lines 125 and 126 to find if the program has entered an incorrect state and throw an exception.

```
if (!"...".equals(previous....()) && lineIndent > prevIndent) {
    if (!":".contains(prevLineLastChar)) {
```

Table 4.2: Statements as categorised through a negotiated agreement. Bold values represent the technique with the highest count (© 2025 IEEE).

Category		Total	CC_{DS}	CC_{OS}	PTSI
Checks	Special Case Check	7	**7**	2	0
	State Check	6	**6**	0	0
	Output Check	4	0	**4**	4
	Trivial Output Check	4	0	**4**	1
	Input Check	1	**1**	0	0
Update	Output Update	15	**15**	12	12
	State Update	6	**6**	0	0
Initialisation	State Initialisation	7	**6**	2	1
	Output Initialisation	5	**5**	0	0
Return	Output Return	9	**6**	0	4
	Default Return	3	1	0	**2**
	Private Method Return	1	**1**	0	0
	Trivial Output Return	1	**1**	**1**	0
Other	Data Loading	322	68	**310**	8
	Iteration	25	**24**	2	1
	Defensive Programming	14	9	**11**	6
	String Processing	12	0	**12**	11
	Mathematical Computation	5	**5**	0	0
	Throw Exception	3	**3**	2	2
	Scheduling	2	**2**	**2**	0
	Parent Call	1	0	**1**	**1**

We identified 4 statements as output checks, where the output was the method's return value. Each instance occurred in the MyStringUtils class, to iteratively perform string processing (and therefore also included later in this section), such as the following ineffectual while statement check on Line 1801.

```
while (text.contains("\b")) {
  text = text.replace("\b", " ");
}
```

Trivial Output Checks included code that, given the set of inputs, applying a given operation was redundant, so an early check was performed to avoid it. In the method "capitalise" in MyStringUtils, for example, if the first character was already a capital, it was redundant to capitalise it, so it returned the string without changes, as shown below:

```
if (Character.isUpperCase(c)) {
  return s;
}
```

The check itself was the ineffectual statement in this example.

We found an instance of an Input Check in MyStringUtils using a try statement that

we could not classify under another category. The code used the ineffectual try statement to attempt to parse an input string as an integer, identified only by CC_{DS}.

Update (21)

We labelled updates as statements that reassigned existing variables, based on the program's behaviour. In particular, we identified State Updates and Output Updates. State updates included class state or internal method state statements, such as updating a counter variable or boolean flags used to track state. All 6 were identified by CC_{DS} as in the oracle gap. One example in the OptimalStringAlignment class tracked the cost while calculating a distance matrix.

```
cost = 1;
if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
    cost = 0;
}
```

Here, CC_{DS} deemed both the cost variable updates ineffectual.

Output updates included reassignments to the variable being returned by the method. CC_{DS} placed all 15 such statements in the gap, but CC_{OS} and PTSI also identified 12 of these.

Initialisation (12)

Initialisations include ineffectual variable declarations. These are generally variable declarations where the initialised value does not impact oracle outcomes. State Initialisation included variables declared solely to track the program state. We found this to occur 7 times, with 6 being identified by Slicer4J, 2 by PORBS and 1 by PseudoSweep. For instance, in the WellIndented class, Line 93 initialised a boolean flag to track "withinBlockScalar" throughout the method:

```
boolean withinBlockScalar = false;
```

The oracle gaps also contained 5 Output Initialisations that initialised the variable returned by the method. For example, the "lowerTriangle" method in the IntSquareMatrix class contained an ineffectual output initialisation on Line 364, as the triangle variable is returned later in the method.

```
IntArray triangle = new IntArray((n * (n + 1)) / 2);
```

Return (14)

We used return as a category, but delved further into the purpose of the return statement. This sample had no void-return statements, so all statements returned a variable or a value. We found one instance where the return statement returned a Trivial Output that did not need further operations from the method. Line 21 in the Rectangle class shows

where, as part of a comparator for ordering, an initial check returns early when the two inputs are equal.

```
    if (o1.equals(o2)) return 0;
```

We also found one instance of a return statement in a Private Method. We labelled this as such because it was related to why it was in the oracle gap. This example was in the OptimalStringAlignment class on Line 120, within an internal implementation of a minimum method as shown below:

```
private static int min(
        final int a, final int b, final int c) {
    return Math.min(a, Math.min(b, c));
}
```

The three Default Returns appeared when the program behaviour had not met any previous checks and therefore reached a specified default return value. An example is Line 1981 in the "hasJapaneseCharacter" method of MyStringUtils, where after iterating through each character in a string, if none of the characters fulfils the check, the method returns false.

```
public static boolean hasJapaneseCharacter(String str) {
  for (char c : str.toCharArray()) {
    if (JAPANESE_BLOCKS.contains(UnicodeBlock.of(c))) {
      return true;
    }
  }
  return false;
}
```

The most prominent return type in the oracle gap was the Output Return, which returned the result computed by the method. There were 9 instances of this, identified between $gap(\mathrm{CC}_{DS})$ with 6 and $gap(\mathrm{PTSI})$ with 4. Below is an example from Line 371 in the IntSquareMatrix class, where the "lowerTriangle" method ineffectually returns said triangle under CC_{DS}.

```
public IntArray lowerTriangle() {
    int n = rows;
    IntArray triangle = new IntArray((n * (n + 1)) / 2);
    int count = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j <= i; j++) {
            triangle.setElementAt(count++, flmat[i][j]);
        }
    }
    return triangle;
}
```

Both PTSI and CC_{OS} deem this statement effectual as they are deletion-based and this statement cannot be deleted or replaced with `null`. However, CC_{DS} states that no test has data or control dependencies on this statement.

Other (384)

The Other category contains the code descriptions that did not fall into clear groupings. We discuss the categories in order of total statements. The largest category in this group was the Data Loading category, containing 322 statements, most of which were

identified by the OGCA using PORBS, as shown in Table 4.2. These statements primarily appeared in the MyStringUtils class, which loaded many strings into map structures, such as Line 158:

```
escapeStrings.put("&amp;", new Character('\u0026'));
```

The next most prominent category under Other was Iteration, which included for loops, while loops and other structures primarily for iteration. Although this category does not provide insight beyond the statement types, it was worth noting that they were revealed mainly by the CC_{DS} approach.

Interestingly, we found 14 Defensive Programming statements in the oracle gaps, with some being revealed by each OGCA. These statements evaluated and addressed inputs and states to ensure that unexpected exceptions would not arise later in the program. For example, the OptimalStringAlignment class throws a NullPointerException if String s1 is null:

```
if (s1 == null) {
    throw new NullPointerException("...");
}
```

String Processing accounted for 12 statements, none of which were identified by CC_{DS}. These included the following in the "removeAccents" method in the MyStringUtils class:

```
s = s.replace((char) 0xE1, 'a');
```

In this instance, the deletion-based techniques of CC_{OS} and PTSI seem tailored to detect string-processing statements that may require further targeting test cases or assertions.

Mathematical Computation includes any statement used to perform a mathematical operation given some values. We found 5 instances of this in the IntSquareMatrix class, where the whole class is directed at mathematical operations on matrices. For example, the following three ineffectual lines are used as part of a transpose matrix operation and were identified only in the oracle gap calculated with Slicer4J.

```
int t = flmat[i][j];
flmat[i][j] = flmat[j][i];
flmat[j][i] = t;
```

The smallest three categories in Other were Throw Exception, Scheduling and Parent Call. The three Throw Exception statements were another example of cases where we could not assign a deeper purpose than the statement type. This does not include all ineffectual exception throws, just those that could not be attributed to another purpose. We found two examples of Scheduling statements that were only identified as being in the oracle gap by CC_{DS} and CC_{OS}. The PseudoSweep-based PTSI was likely unable to identify these due to its difficulties with threaded code. Although this study aimed to avoid threaded code, there may have been instances where our pattern-matching method did not reveal it. We found one instance where a parent call was in the gap, highlighted by CC_{OS} and PTSI. For this, the parent call was for a custom-written equality check on Line 211 in IntSquareMatrix:

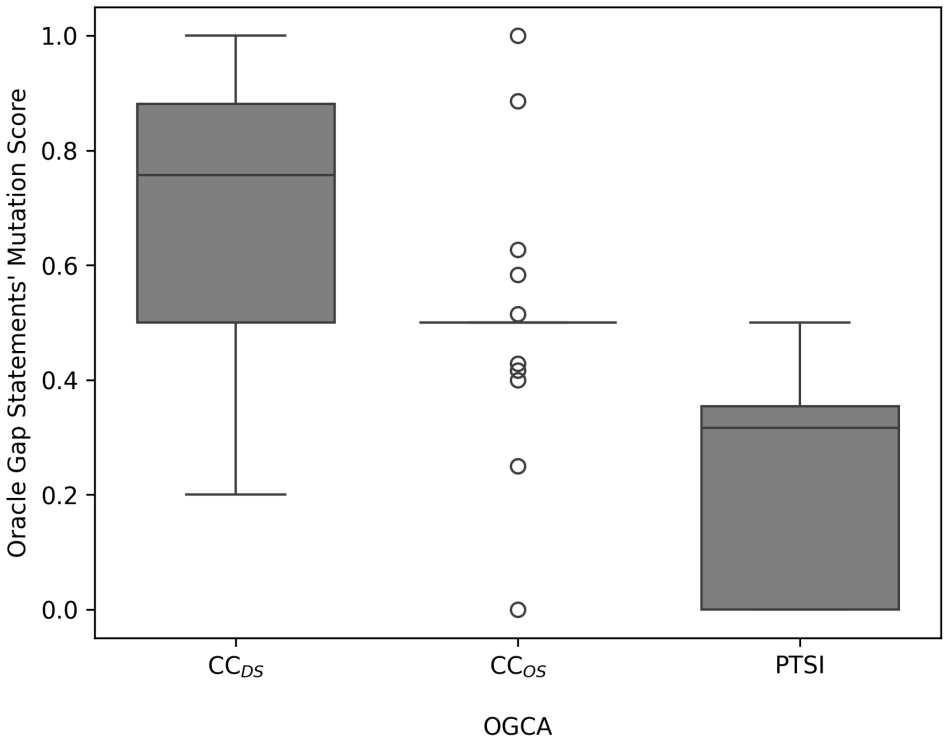

Figure 4.2: A boxplot of the mutation scores for the statements in each oracle gap. Each box represents an interquartile range (IQR). The whiskers (i.e., the lines) extend to the max. and min. values but exclude outliers (i.e., the dots) (© 2025 IEEE).

```
public boolean isEqualTo(IntSquareMatrix r) {
    return super.isEqualTo((IntMatrix) r);
}
```

This class did not invoke Java's built-in equality methods.

Conclusion for RQ3. The gaps created by the three OGCAs included statements from 21 sub-categories across 5 overarching categories. The CC$_{DS}$ gap contained the most control-flow type statements, such as ineffectual checks, iterations and initialisations, whereas CC$_{OS}$ and PTSI highlighted more data-flow type statements such as ineffectual string manipulation, data-loading and defensive programming. The divide between dynamic slicing and the deletion-based approaches suggests the code-under-test should influence OGCA selection.

4.4.4 RQ4: Fault Detection

Figure 4.2 presents the mutation scores for all of the statements within each oracle gap for each class by OGCA. We see the oracle gaps for each class as calculated using CC$_{DS}$, which has the highest median mutation score, with a wide range from 0.20 to 1.00. This suggests that the approach may be less effective at highlighting under-tested areas where these faults may exist. In other words, this oracle gap identifies statements with high

Table 4.3: Mutants in oracle gap statements by mutation operator, and the percentage of each killed (© 2025 IEEE).

	CC_{DS}			CC_{OS}			PTSI		
Mutation Operator	#Killed	#Survived	%MS	#Killed	#Survived	%MS	#Killed	#Survived	%MS
RemoveConditionalMutator_EQUAL_ELSE	47	31	60	17	54	24	4	11	27
RemoveConditionalMutator_EQUAL_IF	42	35	55	50	18	74	6	5	55
MathMutator	51	16	76	17	12	59	0	8	0
ConditionalsBoundaryMutator	28	21	57	11	12	48	6	2	75
RemoveConditionalMutator_ORDER_IF	39	10	80	20	3	87	7	1	88
RemoveConditionalMutator_ORDER_ELSE	32	16	67	5	18	22	0	8	0
VoidMethodCallMutator	10	2	83	2	8	20	0	3	0
NullReturnValsMutator	10	0	100	5	0	100	0	1	0
PrimitiveReturnsMutator	13	0	100	2	0	100	0	1	0
EmptyObjectReturnValsMutator	2	2	50	7	0	100	0	2	0
BooleanTrueReturnValsMutator	1	1	50	2	2	50	3	3	50
IncrementsMutator	4	0	100	1	1	50	0	0	-
BooleanFalseReturnValsMutator	0	0	-	1	2	33	2	0	100
InvertNegsMutator	3	1	75	0	0	-	0	0	-
All Operators	282	135	68	140	130	52	28	45	38

mutation scores, which are therefore a lower priority for further testing. Ultimately, these high scores suggest that addressing the oracle gap identified by dynamic slicing may not help developers to reveal faults in their code. In contrast, PTSI's oracle gap for pseudo-tested statements yields the lowest mutation scores, with a median of 0.32, indicating that it is the most effective of the three approaches at highlighting potential deficiencies in fault detection. The mutation scores are generally low and show little variation, indicating consistency in PTSI's pinpointing of tests that fall short by exhibiting weak fault detection. The mutation score of CC_{OS}'s oracle gap sits between the scores of the other two OGCA's gaps, with the smallest inter-quartile range of 0.00, causing no visible box in Figure 4.2. This means there is no variability in the middle 50% of the data; in this case, the mutation scores were 0.50. Yet, it has the most significant overall spread, with mutation scores at both 0.00 and 1.00. Although less effective than PTSI at highlighting areas of low fault detection, it is the most consistent of the OGCAs.

Table 4.3 shows the PIT mutants placed in statements that appear in each oracle gap. Across all oracle gaps, the conditional logic based operators account for the majority of surviving mutants. However, it becomes apparent that different mutation operators survive in different gaps, pointing to the differing use cases for each gap. In the CC_{DS} gap 135 mutants survivived, compared to 130 for CC_{OS} and 45 for PTSI. For CC_{DS} the conditional operators had the highest counts of surviving mutants. In particular, this included `RemoveConditionalMutator_EQUAL_IF`, `RemoveConditionalMutator_EQUAL_ELSE` and `ConditionalBoundaryMutator` (35, 31 and 21 surviving respectively). Yet, by percentage mutation score, these were among the operators with the highest survival rates. Interestingly, there were no surviving `IncrementMutator` mutants in the dynamic slice-based oracle gap. The CC_{OS} oracle gap also had the `RemoveConditionalMutator_EQUAL_ELSE` as well as `RemoveConditionalMutator_ORDER_ELSE` and `VoidMethodCallMutator` with the highest survival rates (24%, 22% and 20% Mutation Scores respectively). Compared to CC_{DS},

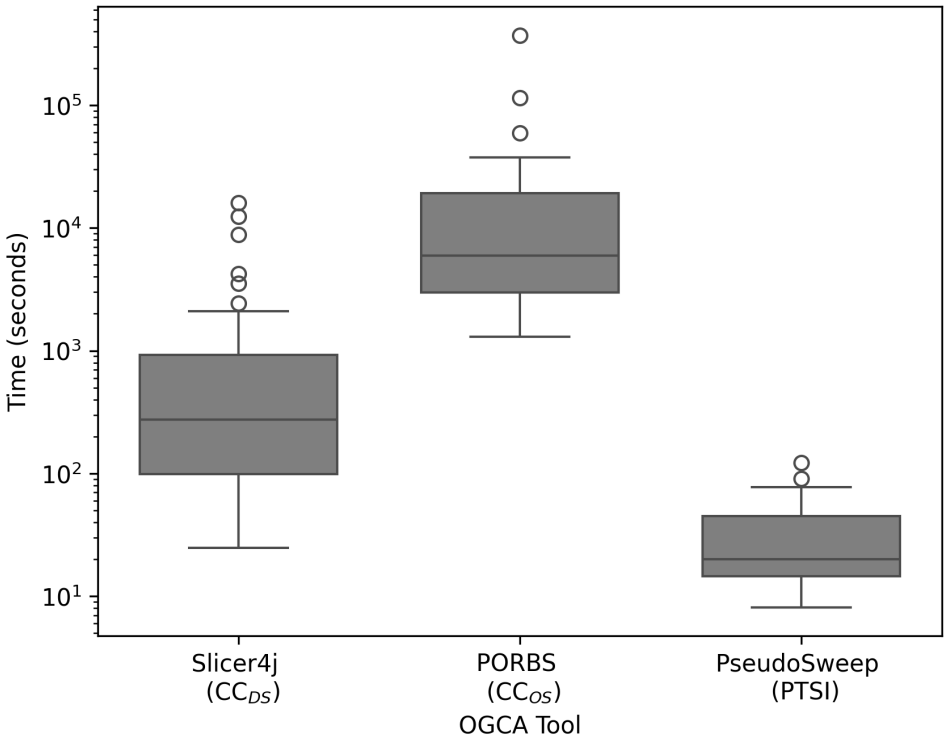

Figure 4.3: A boxplot with a logarithmic scale showing the differences in execution times for the tools implementing CC_{DS}, CC_{DS} and PTSI (© 2025 IEEE).

where the `VoidMethodCallMutator` had a low survival rate, the differences in the types of code appearing in each oracle gap continues to emerge in these results. PTSI, the other deletion-based techique, had a 100% survival rate for the `VoidMethodCallMutator` highlighting the differences in the types of faults that could persist in each gap. PTSI had 6 operators with 100% survival rates (i.e., 0% mutation score), which were likely due to the low numbers of mutants. Notably, 3 of these occurred for operators where CC_{OS} had 0% survival rate. These operators were each `Returns`-based operators highlighting that while the two OGCAs often have overlap, they also have differences and are not interchangable.

Conclusion for RQ4. The oracle gaps calculated by PTSI have the lowest mutation scores, suggesting it is most effective at revealing areas of fault-detection deficiency. The operators that survived in each of the gaps differed and, therefore, shows they cannot be used interchangably.

4.4.5 RQ5: Performance

Figure 4.3 presents a box plot of each OGCA's execution times. As explained in Section 4.3, the tools used by each OGCA are: Slicer4J for CC_{DS}; PORBS for CC_{OS}; and PseudoSweep for PTSI. Due to a significant time difference, Figure 4.3 uses a logarithmic scale. The Slicer4J execution times span from 24.8 seconds to 16067.0 seconds (approx. 4.5hrs). The box ranges from 99.3 seconds to 921.8 seconds (approx. 15 min), with no

overlap with quartiles from other tool execution times. The Slicer4J execution times are generally higher than those of PseudoSweep, but mostly lower than PORBS. Slicer4J also shows more variability in execution time with some high outliers. Overall, PseudoSweep has the lowest execution times, ranging from 8.1 seconds to 121.7 seconds. This is the smallest range of any OGCA, as well as the least variability with an inter-quartile range of 30.2 seconds. We also noted that PseudoSweep had similarly quick execution times to PIT (i.e., the mutation testing tool used to answer RQ4), even though PseudoSweep is still a prototype tool.

PORBS' execution times are significantly higher than the other OGCAs, ranging from 1297.0 seconds (approx. 21min) to 371675.3 seconds (over 4 days) for a single Java class. This results suggests that using PORBS on an entire project becomes infeasible if it contains more than a few classes. The inter-quartile range of 16272.2 seconds (approx. 4.5hrs) for PORBS execution times also exposes its limited practicality.

Conclusion for RQ5. PORBS (CC_{OS}) was consistently impractically slow by a large margin, while Slicer4J (CC_{DS}) had varying execution times. Overall, PseudoSweep (PTSI) consistently had the fastest execution times per class.

4.5 Conclusions

Calculating an oracle gap enables developers to pinpoint covered statements that do not cause an assertion to pass or fail (**B1**), revealing vulnerabilities to incorrect behaviour and highlighting where testing is needed. Yet, with multiple OGCA options, developers cannot make an informed selection. This study quantitatively and qualitatively compared the oracle gaps of CC_{DS}, CC_{OS} and PTSI (**R2**) across 30 Java classes in six projects to assist developers in choosing a suitable OGCA with the key findings as follows:

1. Each OGCA produced dissimilar oracle gaps in both statement set size and content (**B2**).

2. The set sizes ranged from 67–583 ineffectual statements, with the low Jaccard similarity scores (0.06–0.21) indicating that each OGCA identifies largely distinct sets of ineffectual code (**B2**).

3. Each OGCA identified different ineffectual statement types, with CC_{DS}'s gaps frequently involved iteration and update statements, while CC_{OS} and PTSI focused on data loading and string processing(**B3**).

4. *gap*(PTSI) had the lowest mutation scores, making these statements high priority for improved fault detection(**B2**).

5. PseudoSweep for PTSI emerged as the fastest OGCA tool (**B2**).

As such, this study identified PTSI as the most efficient and effective OGCA, enabling developers to quickly highlight the most vulnerable statements that require additional tests and assertions in the shortest execution time. Ultimately, the combination of this chapter and the proposed future work in Section 6.2 will yield a practically useful way for developers to automatically identify how their test cases fall short.

However, while identifying oracle gaps assesses a statement's influence on an assertion, it neglects to consider whether the test suite can actually detect semantically complex faults that a developer may encounter. The recent rise of large language models in software testing has enabled the generation of complex, context-specific transformations that create hard-to-define mutants. Therefore, this thesis presents Chapter 5, which employs LLMs to explore method-level mutants that rule-based mutation may overlook.

Chapter 5

Empirically Comparing Hazard-Guided LLM Mutation with existing techniques

The contents of this chapter are based on the accepted paper "M. Maton, G. M. Kapfhammer, and P. McMinn. Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2026, DOI: 10.1145/3816483.3816524".

Author Contribution For this publication, I was responsible for prototyping the *multiplex* framework used in this study and for integrating each evaluated technique, under guidance and supervision from the second and third authors. I designed and implemented the experimental design, collected data and prepared the manual analysis, which was then undertaken by all authors collectively. The submitted version of this paper was primarily drafted by the second and third authors, with feedback and edits from the first author.

Use in this Chapter For this chapter, I have edited the content to align with the background provided in Chapter 2 and added examples to the discussion. To maintain a consistent voice throughout this thesis, I have independently re-authored all sections drafted by co-authors.

5.1 Introduction

Mutation testing is widely agreed to be dependent on the "competent programmer hypothesis", which assumes developers write code that is close to being correct [48]. Mutation testing, therefore, exploits this assumption by inserting small synthetic faults into the code to mimic the mistakes a developer could make. Recently, studies have found that

existing mutation operator sets are "not representative of real-world faults" and "cannot be reliably used to reproduce these faults" [12]. Studies such as this have each found that while the mutants themselves couple with real faults, the existing operator sets are not extensive enough to produce representative real-world faults, missing operators, for example, that can introduce new method calls and add code blocks [12, 63, 68, 98]. Findings such as these highlight the need for mutation testing tools that can generate context-specific mutants that resemble real-world developer mistakes. Importantly, how can we systematically generate context-aware mutants while ensuring they remain productive?

The initial attempts to address this challenge employed deep learning to train models on a dataset of historical faults. However, the requirement for developers to train a specialised model on real defects makes industry adoption impractical. Furthermore, Ojdanic et al. found that mutants generated in this way — e.g., those from DeepMutation— are both easy to kill and can be easily subsumed by other approaches [144, 188]. More recent work has explored the use of pre-trained language models (PLMs) and large language models (LLMs) to generate context-specific mutants through processing the underlying semantics of the code under test. Tools such as μBERT, LLMorpheus and Mutahunter use approaches such as "masking", which hides small code regions (e.g., single lines or AST nodes) and prompts the language model to predict replacement text [36, 103, 187]. However, since these tools target such small code regions, they cannot reflect the larger "missing operators" identified by the fault coupling studies. BugFarm, on the other hand, directs the LLM to mutate a set of "least attended statements", to create multiple smaller code transformations to produce a single "hard-to-detect" and "hard-to-repair" mutant [85]. While these mutants are harder for a developer to test for, they again do not target realistic developer mistakes that may arise from inexperience or misunderstanding of a program's specification.

To target this, it is essential to enable the LLM to mutate entire methods while providing sufficient constraints to mitigate uncontrolled and unhelpful transformations. The challenge, therefore, is to systematically guide the LLM to produce context-specific mutants that reflect real-world defects without compromising properties such as compilability. To address this, this chapter introduces the use of hazard analysis techniques to interpret code intention and guide LLM-based mutant generation. In safety-critical fields, such as process engineering, methods like HAZOP (Hazard and Operability Study) and STPA (System-Theoretic Process Analysis) are used to systematically review potential risks in process or system design, identifying the possible causes and consequences of things going wrong [108, 119]. Applying HAZOP and STPA to the mutant generation process can therefore guide the LLM to generate mutants that reflect hazardous implementation of the original program through describing the method and applying guide words to generate deviations (HAZOP) and unsafe control actions (STPA).

As outlined in Figure 5.1, this chapter's approach operates in three phases (**C1**). The

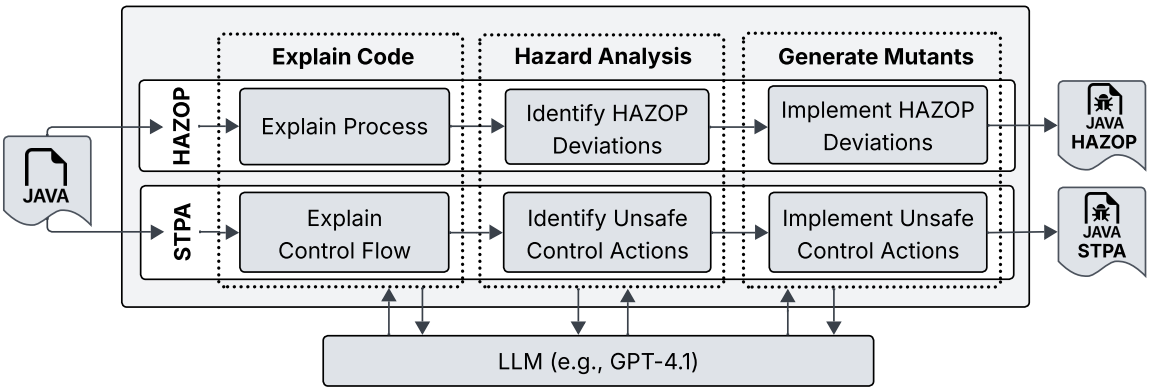

Figure 5.1: Hazard Analysis Mutation Generation approaches using LLMs Overview (as implemented in *multiplex*).

first phase extracts the algorithm as either a list of natural language steps for HAZOP or as a control flow diagram for STPA. The second phase applies hazard analysis guide words to these code explanations to identify potential deviations or unsafe control actions. The third phase re-implements the code, incorporating the deviation or unsafe control action. This approach (described in further detail in Section 5.2) integrates the LLM's capacity for semantic reasoning with systematic hazard analysis frameworks to enable the substantial yet directed changes to the code under test. Unlike LLMorpheus and BugFarm, this approach guides the LLM to systematically implement method-level behavioural changes, to yield diverse and "aggressive" mutants to help developers improve test suites.

This chapter empirically compares HAZOP- and STPA-guided LLM-based mutants with both rule-based and LLM-based approaches (**R3**), assessing their strengths and roles across 15 Java projects in the Defects4J dataset (**C2**). Our experiments (outlined in Section 5.3.3) revealed the following. For **RQ1**, method-level mutants generated without guidance have lower compilability than those generated with other LLM techniques. While hazard analysis improved this, all LLM-based techniques produced uncompilable mutants. For **RQ2**, the volume of mutants may bias the subsumption analysis, making it difficult to evaluate relative mutant strength. After applying random sampling, the unguided technique continued to perform poorly, while the other LLM-based techniques complemented the strong mutants from the rule-based technique. For **RQ3**, the volume of mutants may again bias the defect-matching analysis. After again randomly sampling the mutant sets to control for size, the unguided LLM continues to perform poorly, while the hazard analysis techniques exhibit behaviours that uniquely trigger a significant proportion of defect-finding tests. For **RQ4**, performing each LLM-based technique using a smaller, local LLM yielded similar performance to the cloud model used in RQs1–3, in terms of mutant compilability, subsumption relationships and defect matching. Finally, a qualitative manual analysis revealed that although the hazard-guided mutants can be

syntactically large, they can produce subtle semantic differences undetected by the test suite, raising questions about both test and code quality. Therefore, the contributions of this chapter are as follows:

C1. A Hazard-guided mutation technique, leveraging LLMs to generate code descriptions, apply hazard analysis and mutate Java methods to implement risky behaviours (Section 5.2).

C2. An empirical study to understand the role of hazard-guided mutation relative to state-of-the-art rule-based and LLM-based techniques (Section 5.3).

5.2 Approach

This section introduces our approach, which uses LLMs to perform hazard analysis of the method under test's functionality before generating a mutant that implements a given hazardous behaviour. We implement this approach using HAZOP and STPA in our tool, *multiplex* (*mu*tation analysis with *l*arge language models for *t*esting, *i*n a *p*rompt-*l*evel *ex*perimentation framework), as shown in Figure 5.1. However, *multiplex* also serves as a platform for replicating other LLM-based mutation tools through its modular design.

Our approach, when implemented using either HAZOP or STPA, adopts the following structure. First, as outlined in Figure 5.1, the **(1) Explain Code** step extracts the method source code and prompts the LLM to return a detailed description of the method. Secondly, **(2) Apply Hazard Analysis** takes the method descriptions and applies the hazard analysis techniques to derive possible risky behaviours using a prescribed list of guide words. Finally, the **(3) Generate Mutants** step takes the risky behaviours and implements them as full method mutants. We provide the LLM with the original code, since, when given only the descriptions, it can hallucinate method calls and type information, often leading to uncompilable mutants.

The motivating Java method in Listing 5.2 shows the `isAlphanumeric` method from the Apache commons-lang3 `StringUtils` class. Given a sequence of characters, the method iterates over the sequence, checking each character to ensure it is a letter or a digit. If any character is not a letter or a digit, it returns `false`; otherwise, it returns `true`. We refer back to `isAlphanumeric` as we step through the approach.

5.2.1 HAZOP

The HAZOP process assesses potential risks and hazards arising from deviations from the system's intended functionality and highlights scenarios that may require mitigation [167]. We refer to our HAZOP implementation in *multiplex* as "*m*-HAZOP".

```
1   public static boolean isAlphanumeric(CharSequence cs) {
2      if (cs == null) {
3          return false;
4      }
5      int sz = cs.length();
6  +H  boolean hasDigit = false;
7      for (int i = 0; i < sz; i++) {
8  +S      char c = cs.charAt(i);
9  +S      if (c == '\_' || c == '-') continue;
10         if (Character.isLetterOrDigit(cs.charAt(i)) == false) {
11                 return false;
12         }
13 +H      if (Character.isDigit(c)) {
14 +H          hasDigit = true;
15 +H      }
16       }
17 -H   return true;
18 +H   return hasDigit;
19    }
```

Figure 5.2: The `isAlphanumeric` method from the Apache commons-lang3 `StringUtils` class, shows the original code diffed against a HAZOP and an STPA mutation. Lines marked `+/-` indicate additions and removals; `H/S` highlight *m*-HAZOP or *m*-STPA generated changes.

(1) Explain Process

m-HAZOP begins by using an LLM to describe the method-under-test as a step-by-step process (i.e., its design intent) in a numbered, natural-language list. The "Code Explanation Snippets" in Table 5.1 show examples of code explanations that an LLM has derived from the `isAlphanumeric` method in Figure 5.2. The prompt asks the LLM explicitly to "explain what the code does, not how it is written", to gain the semantic intent of the line, rather than a syntactic description. For example, *m*-HAZOP describes Line 17 as "Return true if all characters are alphanumeric" which captures the behaviour in the context of the whole method, rather than a syntactic description.

(2) Identify HAZOP Deviations

m-HAZOP then prompts the LLM to identify HAZOP deviations, using the guide words shown in Table 5.1, for the line-by-line descriptions. The HAZOP guide words are established standards for evaluating the causes and consequences of risk in a safety critical process [8]. However, they are also applicable to software processes as both domains follow step by step descriptions of inputs, outputs, conditions and execution control as demonstrated by the adaptations to the software domain, including rule-based Java mutation operators [105, 116]. While the HAZOP guide words are considered complete in safety critical processes, it is important that we continue to explore their compeleteness in the software space.

m-HAZOP provides the LLM with clear guidance and structure, as used in process engineering HAZOP applications. This results in a set of deviations from the method

Table 5.1: Applying HAZOP Guide Words. Examples of applying HAZOP to the LLM-extracted code explanation of `isAlphanumeric` (Figure 5.2). In the example deviations, added words appear in **bold**, while removed words appear in ~~strikeout~~.

HAZOP Guide Word	**Code Explanation Snippet** *Example Deviation*
No (not, none)	Return true if all characters are alphanumeric *Return **no** true **value even** if all characters are alphanumeric*
More (more of, higher)	Return false if the input is empty *Return false if the input **length** is ~~empty~~ **greater than a minimal threshold***
Less (less of, lower)	Iterate over each character index in the input *Iterate over ~~each~~ **fewer** character ~~index~~ **indices** in the input*
Other than (other)	Return false if the input is null *Returns ~~false~~ **true** if the input is null*
Part of	Starts a loop to examine each character *Starts a loop ~~to~~ **that** examine**s only some** ~~each~~ character**s***
Reverse	Return true if all characters are alphanumeric *Return true **only** if ~~all~~ characters are **not** alphanumeric*
As well as (more than)	Return true if all characters are alphanumeric *Returns true if all characters are alphanumeric **as well as at least one digit is present***

description that is acting as the method design. In the "Starts a loop to examine each character" explanation example, as shown in Table 5.1, *m*-HAZOP applies the guide word "PART OF", resulting in the deviation "Starts a loop to that examines only some each characters".

(3) Implement HAZOP Deviations

Finally, *m*-HAZOP reimplements the method, incorporating the deviations to break the algorithm away from the original design and create a mutant. For instance, the last deviation in Table 5.1 applies the guide word "AS WELL AS" to require that at least one digit is present for the sequence to be considered alphanumeric. *multiplex* generates one implementation per deviation. Figure 5.2 shows *m*-HAZOP's mutant implementation, annotated using `+/- H`, where it adds an extra check on Lines 13–15 to ensure the sequence fulfils this requirement.

5.2.2 STPA

STPA explores potential hazards in a system's control flow, where individual components may be functionally correct but the control flow leads to incorrect use, as described in Section 2.5.7. We refer to our STPA-based mutation testing implementation as *m*-STPA.

```
digraph ControlFlow {
    node [shape=box];

    start [label="Start"];
    checkEmpty [label="Check if cs is empty"];
    returnFalseEmpty [label="Return false (cs is empty)"];
    computeSize [label="Compute sz = cs.length()"];
    loopInit [label="Initialize i = 0"];
    loopCheck [label="Check if i < sz"];
    charCheck [label="Check if cs.charAt(i) is letter or digit
    "];
    returnFalseInvalid [label="Return false (non-alphanumeric
    char)"];
    increment [label="i++"];
    returnTrue [label="Return true (all chars valid)"];

    start -> checkEmpty;
    checkEmpty -> returnFalseEmpty [label="true"];
    checkEmpty -> computeSize [label="false"];
    computeSize -> loopInit;
    loopInit -> loopCheck;
    loopCheck -> returnTrue [label="false"];
    loopCheck -> charCheck [label="true"];
    charCheck -> returnFalseInvalid [label="false"];
    charCheck -> increment [label="true"];
    increment -> loopCheck;
}
```

Listing 5.1: LLM-generated control flow description using DOT language format.

(1) Explain Control Flow

multiplex starts by prompting the LLM to output "the Java method's control structure into a control flow diagram" using the DOT graph description language [61]. By specifying the diagram representation, the LLM outputs a standardised result for each input program. Listing 5.1 presents an example generated by *m*-STPA for the `isAlphanumeric` method, where the LLM has generated descriptive labels to describe the method's control flow.

(2) Identify Unsafe Control Actions

STPA identifies hazards by examining potential unsafe control actions (UCAs) that controllers (human or software) may take. As described in Section 2.5.7, STPA defines safety as a control problem. Since software is inherently process- and control-based, the STPA guide words apply directly to a control flow description of the software. Recent promise in initial studies has lead to STPA being currently under investigation for use in the aviation and nuclear industries [182, 183]. As such, it has been applied in many software domains, including by Google in a machine learning product development pipeline, making it appropriate to use in the context of mutation testing [169].

Given the control flow diagram description in DOT language, *m*-STPA creates a UCA by applying a guide word to the method control flow, as demonstrated for the `isAlphanumeric` method in Table 5.2. For instance, where *m*-STPA applies the "Too Early" guide word, resulting in the UCA "Return false (non-alphanumeric char) triggered before all chars are checked".

Table 5.2: Applying STPA Unsafe Control Actions. Examples applying STPA guide words to the LLM-extracted code explanation of `isAlphanumeric` (Figure 5.2). In the table, `charCheck`, `loopInit` and `loopCheck` are specific nodes of the DOT graph extracted from the source code of the method and named by the LLM.

UCA Guide word	**Example UCA**
Provided	Return true provides output even if some chars were not checked due to loop error.
Not provided	`charCheck` does not provide correct validation when Character API misclassifies characters.
Too early	Return false (non-alphanumeric char) triggered before all chars are checked.
Too late	Return false (non-alphanumeric char) triggered after invalid data processed.
Out of order	Compute `sz = cs.length()` executed after `loopInit` may misalign loop bounds.
Stopped too soon	Loop terminates before all characters are checked.
Applied for too long	Loop continues beyond string length due to increment or `loopCheck` error.

(3) Implement Unsafe Control Actions

As STPA is control-flow-based, enabling *m*-STPA to reimplement the entire method allows it to implement the complete control-flow changes. *multiplex* generates one implementation per UCA. For the `isAlphanumeric` example in Figure 5.2, *m*-STPA has added Lines 8 and 9 to skip validation of specific characters to implement the "charCheck does not provide correct validation when Character API misclassifies characters" UCA from Table 5.2.

Unlike tools such as LLMorpheus, Mutahunter and Major, *m*-HAZOP and *m*-STPA can mutate the full method in a single mutant, enabling multi-line mutants that other tools cannot produce.

5.3 Evaluation

To evaluate our hazard analysis approach outlined in Section 5.2 and compare with existing LLM- and rule-based approaches, we aim to answer the following research questions:

RQ1: Mutant Properties – How do the mutants generated by our hazard-guided LLM approach compare to those of other techniques with respect to compilability and killability rates, and how similar are they to their original source code?

RQ2: Subsumption Analysis – To what extent does our approach produce redundant, subsumed mutants, and to what extent do the non-subsumed mutants of our approach subsume those of others (and vice versa)

Table 5.3: Large Language Models (LLMs) used in this study.

Model	Type	Training Data Cutoff	Release	Size
GPT-4.1	Closed	Jun 2024	Apr 2025	?
gpt-oss:20B	Open	Jun 2024	Aug 2025	20B

RQ3: Defect Analysis – To what degree do hazard-guided LLM mutants replicate real developer defects? Do they exclusively replicate defects that other approaches do not?

RQ4: Multi-LLM Comparison – How does the performance of hazard-guided LLM mutants vary with different LLMs, compared to other approaches? Can our results be replicated with both local and cloud-based LLMs?

5.3.1 Tools

This section describes the tooling used for this study. As described in Section 5.2, we implemented the HAZOP and STPA techniques into our tool *multiplex* as *m*-HAZOP and *m*-STPA. To enable comparison with other LLM-based mutation approaches, we integrated the prompts from the LLMorpheus and Mutahunter tools (along with the required Java AST parsing) into *multiplex* as *m*-LLMORPHEUS and *m*-MUTAHUNTER, respectively [36, 187]. To provide an LLM-based baseline, we added a further "uninformed" approach that was given the full method and asked to mutate the code without any additional guidance, such as hazard analysis techniques or suggested code changes. To compare and evaluate the hazard-analysis approach against a rule-based mutation technique, we employ the Major mutation testing tool [96]. Major is commonly used in mutation testing research and is integrated into the Defects4J repository for use in research studies [97].

To conduct this study, we executed *multiplex* with both OpenAI's proprietary cloud-based GPT-4.1 and a locally deployed instance of their open-weight gpt-oss:20B model [147] as shown in Table 5.3. As gpt-oss:20B has similar performance to "delivers similar results to OpenAI GPT-o3-mini on common benchmarks", the purpose of using these two models is to assess the *multiplex* implemented techniques' reproducibility, particularly when run locally on developer workstations, rather than a direct comparison [146].

5.3.2 Projects

Since this study evaluates the hazard analysis of processes, we restricted the scope of the evaluation to Defects4J (version 3.0.1) bugs in constructors and methods only (excluding, for example, bugs in the fields of classes). This study does not consider versions in which the tests failed to run or Major did not execute, thereby excluding two complete projects (*Closure* and *JacksonDatabind*). We removed a known flaky unit test

Table 5.4: DEFECTS4J Subjects used in the Empirical Study

Project	Active Bugs in Defects4J	Bugs Evaluated	
		GPT-4.1	gpt-oss:20B
Chart	26	24	19
Cli	39	27	15
Codec	18	17	6
Collections	28	23	20
Compress	47	33	13
Csv	16	15	11
Gson	18	14	9
JacksonCore	26	21	10
JacksonXml	110	5	1
Jsoup	93	26	13
JxPath	22	18	9
Lang	61	12	0
Math	106	24	7
Mockito	38	5	4
Time	26	15	9

(`BugsTest::test13666`) from *Cli* to avoid impacting results. Due to the disproportionately large number of versions in *jsoup* and *math* (93 and 106, respectively), we limit our evaluation to the first 30 bugs to maintain a representative dataset with the resource constraints. These constraints reduced the set under test to 322 bugs. We generated mutants for each of the fixed methods of the originally buggy class for each project in the benchmark. Where non-deterministic aspects of interacting with LLMs caused *multiplex* to fail to generate mutants (e.g., rate limits and timeouts), *multiplex* retries the process once more, keeping only the bugs for which each approach completed successfully. This resulted in 279 bugs for GPT-4.1 and 146 for gpt-oss:20B, as shown in Table 5.4.

5.3.3 Method

To answer the research questions, we execute the *m*-HAZOP and *m*-STPA approaches, along with *m*-LLMORPHEUS, *m*-MUTAHUNTER and *m*-UNGUIDED, on the DEFECTS4J methods described in Section 5.3.2. This produces a set of mutants and their testing outcomes: uncompilable, killed (by which tests), or surviving. We also run the MAJOR mutation testing tool on each subject to generate a set of rule-based mutants and their killmaps. This data enables us to implement the following research method.

RQ1 (Mutant Properties)

To answer RQ1, we quantify the number of mutants generated by each approach implemented in *multiplex*, as well as rule-based MAJOR for further context. We then assess the validity of the generated mutants by measuring the proportion that successfully compile, and, of those, the proportion killed by the original developer test suite. Furthermore,

we use the Levenshtein Edit Distance to compare the mutation sizes of each *multiplex* generated mutant and the original code. This is calculated by ignoring white space (external to string literals) and code comments, as LLMs often add comments and reformat code without making syntactic changes. Using Levenshtein Edit Distance can give us an intuitive sense of the scale of the raw textual changes introduced by each approach.

RQ2 (Subsumption Analysis)

Using our purpose-built scripts, we identify all subsuming mutants from a single approach and calculate the minimal set. That is, for each mutant $m_1 \in M$, there exists no other mutant $m_2 \in M$ such that m_2 subsumes m_1; that is, there is no m_2 where the set of tests killing m_2 is a subset of the tests killing m_1. Therefore, if m_1 is killed by a subset of the tests that kill m_2, m_1 is harder to kill than m_2. As such, the minimal set is the set of mutants that are not subsumed by any other mutant. By identifying the minimal set for each approach, we can assess the redundancy in the mutants it produces and identify its strongest mutants.

Next, we create a combined superset of all the minimal mutants from each approach and calculate the minimal set across all approaches. We then remove any mutants that are killed by the same test set, to keep only the uniquely subsuming mutants, that is, mutants killed by a combination of tests that do not kill any other mutant. This enables comparison of relative mutant strength between techniques. It is worth noting that only killed mutants can be used in subsumption calculations, as killing tests determine the relationships.

RQ3 (Defect Analysis)

To answer RQ3, we analysed the killing test sets for each mutant generated by each approach in *multiplex* and MAJOR and identified those that matched the defect-triggering tests. As such, we could then filter the mutants that uniquely matched the defect-triggering tests. These mutants have therefore uniquely replicated the behaviour (as far as the test suite is concerned) of the real defect in the DEFECTS4J dataset.

For both RQ2 and RQ3, our preliminary experiments suggested that an approach's mutant count may influence the results. To control for this, we employed smallest set downsampling , in which, for each method-under-test (MUT), we identified the smallest mutant set size $n_{\min}$ across all approaches and randomly downsampled larger sets to match it. To address the potential bias of the smallest set remaining unchanged, we also performed half smallest set downsampling by sampling all techniques to $\lceil n_{\min}/2 \rceil$ mutants. For this, we excluded any MUTs with $n_{\min} = 1$, as maintaining a set size of one would fail to mitigate the aforementioned bias. We repeated the random sampling 1000 times for both smallest set downsampling and half smallest set downsampling.

Table 5.5: Mutant Statistics. "Mutants" is the number of mutants generated for each mode; "Compilable" reflects mutants that are syntactically valid; "Killed" represents mutants killed by developer test suites (where the associated perctantage score reflects "kill rate" of compilable mutants).

	Mutants	**Compilable**		**Killed**	
	#	#	(%)	#	(%)
m-HAZOP	3589	3198	(89.1)	2435	(76.1)
m-STPA	2490	2354	(94.5)	1530	(65.0)
m-UNGUIDED	2681	2560	(95.5)	2402	(93.8)
m-LLMORPHEUS	26268	18053	(68.7)	11014	(61.0)
m-MUTAHUNTER	1259	1179	(93.6)	856	(72.6)
MAJOR	24325	24325	(100.0)	15215	(62.5)

RQ4 (Multi-LLM Comparison)

To answer RQs 1–3, we utilise GPT-4.1. To answer RQ4, we re-perform the methods for RQs 1–3 but using a locally deployed LLM, gpt-oss:20B. We then compare the results obtained with each LLM to better understand replicability, particularly using developer-workstation-deployable models.

5.4 Results

5.4.1 RQ1 (Mutant Properties)

Table 5.5 reports the total number of mutants generated by each technique, alongside their compilation rates and the proportion of compilable mutants killed by the developer test suite. *m*-LLMORPHEUS and MAJOR generate significantly more mutants than the other techniques. It is worth noting that MAJOR operates directly at the Java bytecode level; it is designed to produce 100.0% compilable mutants, whereas the LLM-based approaches all produce uncompilable mutants.

However, while most *multiplex* implemented techniques are generally around 90% compilable, *m*-LLMORPHEUS stands out with a significantly lower compilation rate of 68.7 %. As described in Section 2.5.4, LLMORPHEUS operates on the AST, asking the LLM to suggest three possible replacements for specified nodes (as implemented in *m*-LLMORPHEUS) and then adds those code changes into the existing code. The other *multiplex* implemented techniques, however, ask the LLM to mutate whole lines/methods of code directly, giving it greater scope to produce a compilable mutant.

Figure 5.3 presents the distributions of mutation sizes produced by each technique, measured using the Levenshtein edit distance and depicted on a log scale. The figure does not include MAJOR, as the mutants are generated at the bytecode level; as such, not all of its mutants can be represented as source code.

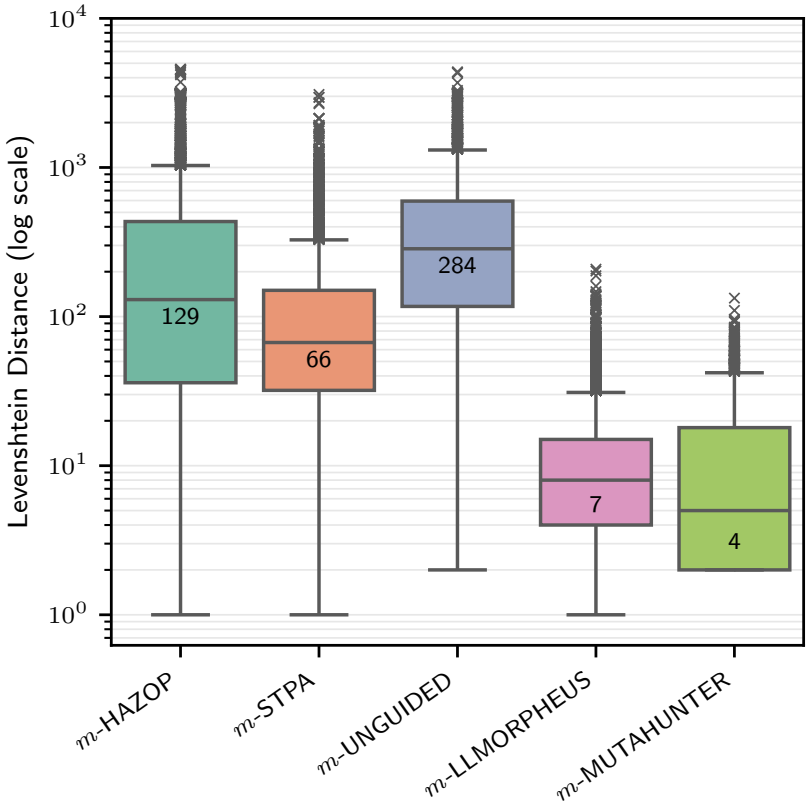

Figure 5.3: Distributions of mutation sizes produced by each technique, measured using the Levenshtein edit distance, depicted on a log scale.

The hazard analysis-guided techniques, *m*-HAZOP and *m*-STPA, generate some of the largest mutants, with median Levenshtein edit distances of 129 and 66, respectively. *m*-UNGUIDED had the highest median edit distance of 284, and with the highest killability, as shown in Table 5.5, at 93.8%, it appears that such large mutations are trivial to kill. Yet despite also being larger, the hazard-guided mutations appear less trivial to kill at 76.1% (*m*-HAZOP) and 65.0% (*m*-STPA). These kill rates are similar to the other techniques, both LLM- and rule-based. LLMORPHEUS and MUTAHUNTER are both limited to mutating up to a line of code, resulting in more minor changes; however, their kill rates remain comparable with all other techniques at 61.0 and 72.6, respectively. As such, the combination of the data in Table 5.5 and Figure 5.3 shows the importance of guiding the LLM towards mutants, as large mutants produced without guidance (i.e., those from *m*-UNGUIDED) can be trivial to kill.

Table 5.6 shows the mutant statistics for *m*-HAZOP and *m*-STPA broken down by guide word. Where the link back to a guide word could not be made due to missing LLM output, the mutant guide word is labelled *Unrecoverabe*. In terms of guide words used by *m*-HAZOP and *m*-STPA, the compilability rates remainded generally consistent. For *m*-HAZOP, most guide words had around 90% compilability, execpt for *As well as* and *Other than* which had compilation rates of 83.5% and 81.9% respectively. These two approaches also had

Table 5.6: Mutant Statistics by Guide Word with GPT-4.1

	Mutants	**Compilable**		**Killed**	
Guide Word	#	#	(%)	#	(%)
m-HAZOP					
No	682	643	(94.3)	491	(76.4)
More	527	462	(87.7)	321	(69.5)
Less	560	508	(90.7)	388	(76.4)
As well as	285	238	(83.5)	163	(68.5)
Part of	386	357	(92.5)	284	(79.6)
Reverse	556	497	(89.4)	407	(81.9)
Other than	540	442	(81.9)	337	(76.2)
Unrecoverable	53	51	(96.2)	44	(86.3)
Total	3589	3198	(89.1)	2435	(76.1)
m-STPA					
provides	545	519	(95.2)	367	(70.7)
does not provide	634	604	(95.3)	416	(68.9)
too early	255	241	(94.5)	149	(61.8)
too late	283	259	(91.5)	136	(52.5)
out of order	291	272	(93.5)	149	(54.8)
stopped too soon	278	263	(94.6)	190	(72.2)
applied for too long	198	190	(96.0)	118	(62.1)
Unrecoverable	6	6	(100.0)	5	(83.3)
Total	2490	2354	(94.5)	1530	(65.0)

lower kill rates, which are calculated as a percentage of the compilable mutants. The *Reverse* guide word had the highest kill rate suggesting the mutants it lead to were the most trivial to kill. For *m*-STPA, all guide words resulted in mutants that were over 91% compilable. However, the killability of the mutants varied significantly more. The lowest kill rate was 52.5% for the *too late* guide word suggesting this guide word leads to harder to kill mutants compare to others. The *out of order* guide word also resulted in mutants with a low kill rate of 54.7%. Compared to *m*-HAZOP, the *m*-STPA guide words tended to produce mutants that were harder kill, however, this was not the case for all guide words. The *stopped too soon* guide word, for example, had the highest kill rate of 72.2% which is comparable with some of the lower kill rates from *m*-HAZOP guide words, such as *More* and *As well as*. Generally, differerent guide words resulted in different kill rates, but none were as trivial to kill as *m*-UNGUIDED mutants.

Furthermore, the boxplot in Figure 5.4 shows the number of lines changed by each approach. As expected due to their designs, *m*-LLMORPHEUS and *m*-MUTAHUNTER each edited a median of 1 line per method, compared to the method-level approaches. Similar to the Levenshtein plot, we can see that *m*-STPA had a lower median of 5 lines edited per method compared to *m*-HAZOP's which edited a median of 9 lines per method. *m*-HAZOP also had a larger inter-quartile range of lines edited, similar to *m*-UNGUIDED. However, *m*-UNGUIDED had the highest median lines edited per method, at 14 but with

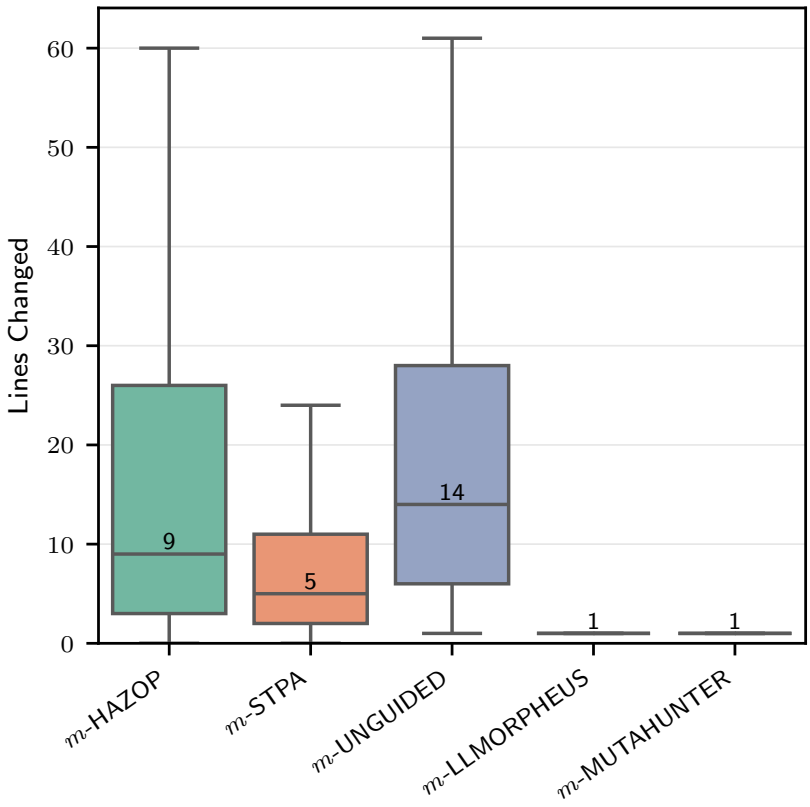

Figure 5.4: Distributions of the number of lines each technique's mutants change in the method under test with GPT-4.1, counted after removing comments and blank lines.

a slightly smaller interquartile range than m-HAZOP. These trends align with the trends seen across the Levenshtein distances, with m-STPA consistently changing fewer lines and m-UNGUIDED changing the most.

Conclusion for RQ1. The compilability and killability of hazard-based mutants are comparable to those of other LLM-based techniques. However, the exception of m-UNGUIDED, which produced large, trivially killable mutants, highlights the need to provide the LLM with guidance for method-level mutation. The different guide words resulted in different compilability and kill rates but none individually resulted in mutants as trivial to kill as m-UNGUIDED.

5.4.2 RQ2 (Subsumption Analysis)

Table 5.7 shows the subsumption analysis results for all mutants killed by developer tests for each technique.

The minimal sets produced in this analysis reveal redundancy across all techniques under test. Where a minimal set forms only a small proportion of the overall set of killed mutants (i.e., % kill. in Table 5.7), this shows a high proportion of the mutants are subsumed, meaning the majority of the mutants generated may be unnecessary.

Table 5.7: Subsumption Analysis

	Total	**Killed**	**Minimal Set**		**Unique Set**	
	# / μ	# / μ	# / μ	% kill.	# / μ	% uniq.
No Sampling (raw counts)						
m-HAZOP	3589	2435	533	21.89	42	3.79
m-STPA	2490	1530	504	32.94	18	1.63
m-UNGUIDED	2681	2402	391	16.28	7	0.63
m-LLMORPHEUS	26268	11014	821	7.46	211	19.06
m-MUTAHUNTER	1259	856	430	50.23	11	0.99
MAJOR	24325	15215	1441	9.47	818	73.89
Smallest Set Downsampling (mean averages)						
m-HAZOP	730.00	627.26	313.65	50.01	56.29	12.63
m-STPA	730.00	685.80	357.05	52.06	79.26	17.78
m-UNGUIDED	730.00	699.80	286.58	40.95	8.45	1.90
m-LLMORPHEUS	730.00	472.24	281.79	59.69	65.01	14.58
m-MUTAHUNTER	730.00	669.54	358.37	53.52	84.27	18.91
MAJOR	730.00	542.35	345.88	63.78	152.41	34.20
Half Smallest Set Downsampling (mean averages)						
m-HAZOP	370.00	318.39	221.78	69.66	46.70	12.94
m-STPA	370.00	347.01	250.47	72.18	72.46	20.08
m-UNGUIDED	370.00	355.90	222.61	62.55	11.44	3.17
m-LLMORPHEUS	370.00	240.25	186.54	77.66	53.90	14.93
m-MUTAHUNTER	370.00	338.15	244.23	72.23	72.01	19.96
MAJOR	370.00	275.35	219.50	79.73	104.40	28.93

For this subject set, under "No Sampling" in Table 5.7, *m*-LLMORPHEUS and MAJOR, while being the highest mutant producers, also have the highest percentage of redundancy, with only 7.46% and 9.47% in their minimal sets, respectively. Conversely, *m*-MUTAHUNTER produces the lowest number of mutants and has the least mutant redundancy. After applying both the smallest set downsampling and half smallest set downsampling measures, the trend changes, with each technique falling within a range of up to 15 percentage points. In both instances, MAJOR had the least redundancy while *m*-UNGUIDED had the most.

The Unique Sets shown in Table 5.7 identify the mutants that, when a superset is created of all the minimal mutants, are uniquely subsuming. That is, which technique's mutants uniquely subsume other techniques' mutants overall. As with the minimal sets, we can see under the raw counts that mutant volume predicts which mutants are uniquely subsuming. After applying smallest set downsampling and half smallest set downsampling, we see that *m*-UNGUIDED creates "weaker" mutants than other techniques, even after balancing the number of mutants. The rule-based MAJOR mutants continue to dominate, producing the highest proportion of subsuming mutants, whereas *m*-LLMORPHEUS becomes one of the weaker mutant producers. Of the LLM-based techniques, *m*-MUTAHUNTER and *m*-STPA contribute the most, uniquely subsuming mutants under smallest set downsampling and half smallest set downsampling, respectively. Note

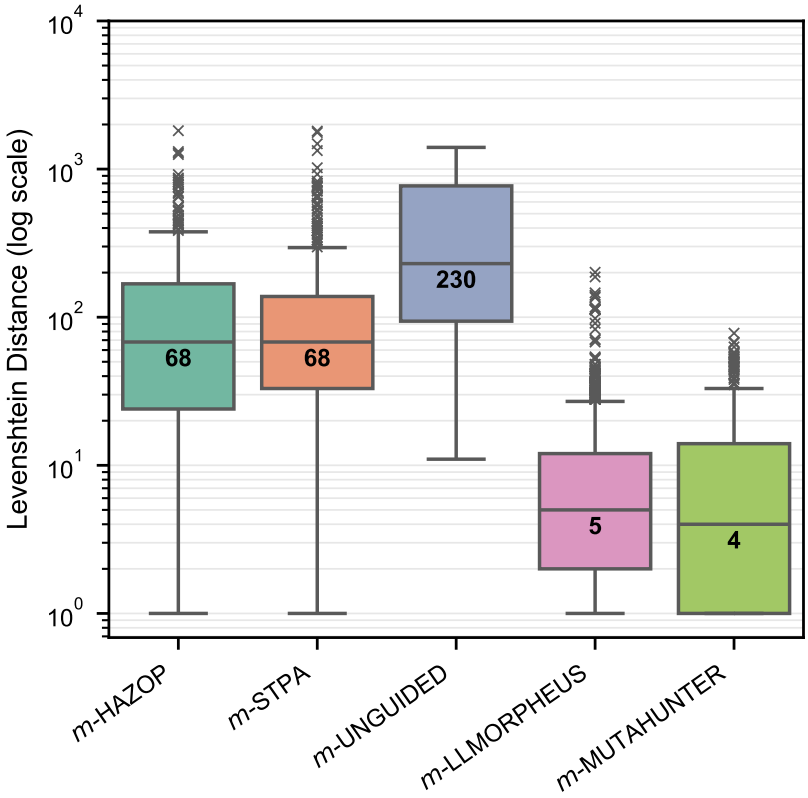

Figure 5.5: Distributions of mutation sizes for uniquely subsuming mutants with smallest set downsampling produced by each technique, measured using the Levenshtein edit distance, depicted on a log scale.

that all techniques contribute some uniquely subsuming mutants in each case, suggesting a complementary nature between the mutant sets.

Figure 5.5 shows that *m*-HAZOP and *m*-STPA— as well as the weaker *m*-UNGUIDED mutants — contribute larger in size uniquely subsuming mutants to the overall set than *m*-LLMORPHEUS and *m*-MUTAHUNTER. *m*-HAZOP and *m*-STPA also contribute a significant proportion of uniquely subsuming mutants to the sampled sets, showing that mutants of these larger sizes have a role to play in producing strong and subsuming mutants.

Table 5.8 shows the unsampled subsumption results for *m*-HAZOP and *m*-STPA broken down by guide word. Where the link back to a guide word could not be made due to missing LLM output, the mutant guide word is labelled *Unrecoverabe*. For *m*-HAZOP, the *No* guide word contributed the most mutants to the minimal set, at 135 mutants. In comparison, the lowest contribution was 30 mutants from the *As well as* guide word. Interestingly, this trend does not continue in their contribution to the unique set. In the Unique Set, the *Less* guide word is most prominent making up 31% of *m*-HAZOP's contribution. Despite contributing the most to the minimal set, the *No* guide word only contributes 14.3% of *m*-HAZOP's mutants to the unique set.

On the otherhand, *m*-STPA's minimal set was dominated by mutants from *provides* and

Table 5.8: Guide Words of Subsuming Mutants with GPT-4.1

	Killed	**Minimal Set**		**Unique Set**	
Guide Word	#	#	% min.	#	% uniq.
No Sampling (raw counts)					
m-HAZOP					
No	491	135	(25.3)	6	(14.3)
More	321	60	(11.3)	4	(9.5)
Less	388	90	(16.9)	13	(31.0)
As well as	163	34	(6.4)	1	(2.4)
Part of	284	60	(11.3)	7	(16.7)
Reverse	407	82	(15.4)	4	(9.5)
Other than	337	66	(12.4)	7	(16.7)
Unrecoverable	44	6	(1.1)	0	(0.0)
Total	2435	533		42	
m-STPA					
provides	367	108	(21.4)	2	(11.1)
does not provide	416	171	(33.9)	5	(27.8)
too early	149	40	(7.9)	4	(22.2)
too late	136	47	(9.3)	3	(16.7)
out of order	149	38	(7.5)	0	(0.0)
stopped too soon	190	51	(10.1)	2	(11.1)
applied for too long	118	49	(9.7)	2	(11.1)
Unrecoverable	5	0	(0.0)	0	(0.0)
Total	1530	504		18	

does not provide, making up 21.4% and 33.9% of the minimal set respectively. While the overall minimal set size was similar to *m*-HAZOP, *m*-STPA's contribution to the unique set was significantly lower at 18 mutants compared to *m*-HAZOP's 42 mutants. Half of these mutants were split between *does not provide* and *too early*, at 5 and 4 mutants respectively. The *out of order* guide word was the only guide word to contribute 0 mutants to the unique set. These results have shown that the guide words not only do not contribute equally subsuming mutants within the minimal sets, but also have different strengths when compared to other tools.

Conclusion for RQ2. The volume of mutants produced by a technique can increase the likelihood that it produces the most subsuming mutants. Applying sampling revealed that *m*-UNGUIDED produced the fewest uniquely subsuming mutants, whereas MAJOR continued to create the most. However, the other LLM-based techniques also contributed significant numbers of uniquely subsuming mutants, including *m*-HAZOP and *m*-STPA, which produced larger mutants, highlighting a complementary role for these mutants in mutation analysis. When broken down by guide word, the minimal and unique sets are contructed unevenly across the guide words, highlighting that some guide words produced stronger mutants.

Table 5.9: Defect Matching

	Total	**Killed**	**Matched**	**Unique**
No Sampling (raw counts)				
m-HAZOP	3589	2435	101	1
m-STPA	2490	1530	106	1
m-UNGUIDED	2681	2402	52	1
m-LLMORPHEUS	26268	11014	144	16
m-MUTAHUNTER	1259	856	86	2
MAJOR	24325	15215	146	11
Smallest Set Downsampling (mean averages)				
m-HAZOP	1029.00	719.10	50.85	4.98
m-STPA	1029.00	655.05	62.80	7.09
m-UNGUIDED	1029.00	948.25	30.46	1.45
m-LLMORPHEUS	1029.00	515.88	47.67	7.15
m-MUTAHUNTER	1029.00	696.64	68.68	8.53
MAJOR	1029.00	714.48	60.62	10.42
Half Smallest Set Downsampling (mean averages)				
m-HAZOP	503.00	354.80	31.00	5.60
m-STPA	503.00	318.60	38.80	7.80
m-UNGUIDED	503.00	459.80	22.40	1.80
m-LLMORPHEUS	503.00	236.40	26.80	5.80
m-MUTAHUNTER	503.00	339.80	39.20	10.40
MAJOR	503.00	344.40	35.80	9.80

5.4.3 RQ3 (Defect Analysis)

Table 5.9 shows the mutants whose killing test sets "matched" the triggering test sets of the defect within that method. Each technique matched at least one defect that no other technique did, with all matching at least 50 defects overall. As in RQ2, the high-volume mutant producers (*m*-LLMORPHEUS and MAJOR) performed well, with *m*-LLMORPHEUS and MAJOR uniquely matching 16 and 11 defects, respectively. The sampling results show this was due to the volume, as with smallest set downsampling, more defects are uniquely matched by other techniques, with these trends being similar in half smallest set downsampling. *m*-UNGUIDED is the only exception to this, remaining much lower than the other techniques even after sampling is applied. Figure 5.6 shows the mutant sizes for smallest set downsampling again showing the *m*-HAZOP and *m*-STPA mutants are significantly larger than the *m*-LLMORPHEUS and *m*-MUTAHUNTER unique defect finding mutants. This trend continues from RQ1 and RQ2, showing that *m*-HAZOP and *m*-STPA are among the largest mutants, each uniquely identifying defects and highlighting their complementary roles in finding real defects.

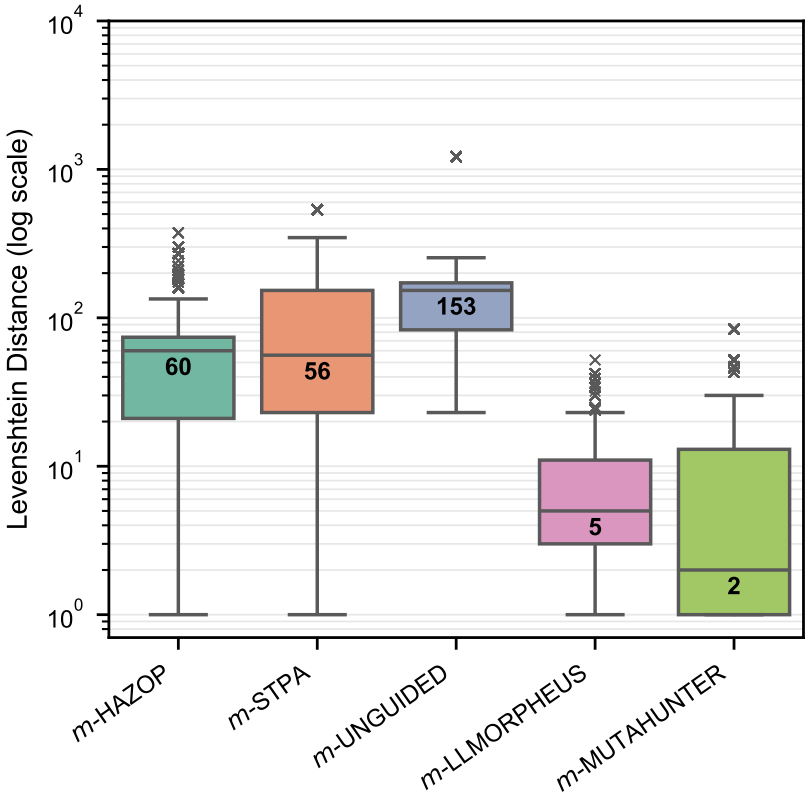

Figure 5.6: Distributions of mutation sizes of mutants produced by each technique that uniquely matching defects, applying smallest set downsampling. Sizes are measured using Levenshtein edit distance, and depicted on a log scale.

Conclusion for RQ3. Without sampling, the results of the defect analysis can be biased by the techniques that generate the most mutants. However, when sampling is used to control for set size, *m*-UNGUIDED continues to perform poorly, while *m*-HAZOP and *m*-STPA identify a significant proportion of unique defects. Given the larger mutants generated by these techniques, this provides further evidence that they play a role in LLM mutation when provided sufficient guidance.

5.4.4 RQ4 (Multi-LLM Comparison)

When generating mutants with gpt-oss:20B, we encountered issues with the context required by the LLMorpheus prompt, leading to context window overflows on multiple occasions. As such, we were only able to collate a complete set of technique results for 146 from the Defects4J dataset.

Table 5.10 shows the mutant statistics for mutants generated by gpt-oss:20B, using the method used to answer RQ1. Interesting, *m*-HAZOP tended to produce more mutants per bug under gpt-oss:20B, whereas *m*-STPA produced significantly fewer. Since STPA is a control-flow-based technique, the volume of UCAs is dependent on the control-flow

Table 5.10: Mutant Statistics with gpt-oss:20B

	Mutants	**Compilable**		**Killed**	
	#	#	(%)	#	(%)
m-HAZOP	4030	3400	(84.4)	2559	(75.3)
m-STPA	612	558	(91.2)	292	(52.3)
m-UNGUIDED	1457	1340	(92.0)	1116	(83.3)
m-LLMORPHEUS	13617	6380	(46.9)	3447	(54.0)
m-MUTAHUNTER	685	628	(91.7)	447	(71.2)
MAJOR	9643	9643	(100)	5210	(54.0)

Table 5.11: Subsumption Analysis (*multiplex* with gpt-oss:20B)

	Total	**Killed**	**Minimal Set**		**Unique Set**	
	# / μ	# / μ	# / μ	% kill.	# / μ	% uniq.
No Sampling (raw counts)						
m-HAZOP	4026	2553	322	12.61	23	5.15
m-STPA	621	296	162	54.73	9	2.01
m-UNGUIDED	1457	1115	228	20.45	9	2.01
m-LLMORPHEUS	13617	3447	369	10.70	78	17.45
m-MUTAHUNTER	682	446	222	49.78	4	0.89
MAJOR	9643	5210	614	11.79	324	72.48

complexity. We noted that the DOT language control flow descriptions *m*-STPA produced using gpt-oss:20B were less descriptive, tending towards syntactic labels rather than semantic descriptions. As such, *m*-STPA generated fewer UCAs, resulting in fewer mutants. Overall, however, the trends in compilability and mutants killed are similar between gpt-oss:20B and GPT-4.1 (i.e., when compared with Table 5.5). We also observed *m*-HAZOP and *m*-STPA, generated mutants of similar size using gpt-oss:20B with median Levenshtein edit distances of 103 and 62, respectively, compared to 129 and 66 with GPT-4.1. These trends continue across both subsumption analysis and defect matching. Table 5.11 shows the subsumption analysis results for mutants generated by gpt-oss:20B. When compared with Table 5.7, MAJOR continues to contribute the highest proportion of subsuming mutants. Both *m*-HAZOP and *m*-STPA slightly increase their share of the uniquely subsuming set with mutants generated using gpt-oss:20B. Table 5.12 also shows similar results for the defects analysis using gpt-oss:20B (without sampling applied) as found in Table 5.9.

Conclusion for RQ4. Using a local model (i.e., gpt-oss:20B) affected the hazard analysis mutant volume; however, other trends, such as compilability, kill rate and size, remained similar to those obtained with a cloud-based model (i.e., GPT-4.1). Mutant subsumption and defect-matching relationships also followed consistent patterns, irrespective of the model used.

Table 5.12: Defect Matching (*multiplex* with gpt-oss:20B)

	Total	**Killed**	**Matched**	**Unique**
No Sampling (raw counts)				
m-HAZOP	4026	2553	67	2
m-STPA	621	296	40	1
m-UNGUIDED	1457	1115	57	0
m-LLMORPHEUS	13617	3447	77	9
m-MUTAHUNTER	682	446	45	1
MAJOR	9643	5210	77	4

5.5 Discussion and Further Observations

Given that *m*-HAZOP and *m*-STPA mutation sizes are generally larger than those produced by other LLM-based and rule-based techniques, the mutants themselves can take quite a different form. This section, therefore, explores a manual analysis of killed and unkilled mutants generated using *m*-HAZOP and *m*-STPA. Although they also had large mutation sizes, we did not consider the *m*-UNGUIDED mutants for manual analysis, as in the results for RQ1–3 they appeared to be trivial to kill, easily subsumed and not as comparable to real faults, reducing their potential use to developers [161].

This section begins by analysing unkilled mutants for equivalence and to determine whether they are productive mutants. As equivalence is a generally undecidable problem [115], we must perform this analysis by hand. We then manually study the *m*-HAZOP and *m*-STPA mutants that uniquely match defects in the results of RQ3 to understand their characteristics. Finally, we study a selection of killed and unkilled (but non-equivalent) mutants to further explore their characteristics, with a particular focus on the unkilled but non-equivalent mutants as they are most likely to be of use to developers.

5.5.1 Unproductive and Equivalent Mutants

We selected six DEFECTS4J projects that varied in domain (*Chart*, *Cli*, *Lang*, *Math*, *Mockito* and *Time*) and randomly selected a subset of five defects for each project. Across these five defects, we sampled five surviving mutants for each technique. There is an exception due to the selected Mockito defects having only three surviving *m*-HAZOP mutants, resulting in an increase in *Cli* to 7 mutants to maintain a total of 60 mutants. Two authors then individually assessed each mutant for equivalence and productivity before meeting to discuss differences and agree on a final consensus. The authors identified 5 of the 30 (16.7%) *m*-HAZOP-generated surviving mutants as functionally equivalent to the original method, along with 15 of the 30 (50%) *m*-STPA surviving mutants. Table 5.5 shows that developer tests killed 2435 of the 3589 *m*-HAZOP mutants, and 1530 of the 2490 *m*-STPA mutants, producing mutant survival rates of 21.3% and 33.1%, respectively. As such, we can estimate the equivalent mutant rate (EMR) (see Equation 2.3) for *m*-HAZOP and

m-STPA to be 3.55% and 16.5%, respectively.

Wang et al. performed a similar scale manual analysis of equivalent mutants, studying rule-based, pre-trained and LLM-based approaches. They found MAJOR to have an EMR of 2.1% and LLMORPHEUS to have an EMR ranging from 3.1–4.1% (dependent on the LLM used). Given our analysis, *m*-HAZOP has a similar rate to LLMORPHEUS, whereas *m*-STPA's rate is considerably higher. However, it remains important to note that, while having a higher EMR, *m*-STPA often outperformed *m*-HAZOP and *m*-LLMORPHEUS in downsampled subsumption analysis and defect analysis, highlighting that, despite its added redundancy, it still produces strong mutants. Our findings also support rule-based mutants having a lower EMR than LLM-generated mutants.

There was one non-equivalent mutant that we deemed "unproductive". The mutant, generated by *m*-HAZOP for *Chart*, added the following statement into a catch block.

```
+H   System.err.println("Error when adding cloned item at index "
+H                           + index + ": " + ex.getMessage());
```

This statement prints a logging statement to the `System.err` output. Such a mutant would not only be difficult for a developer to test for, but would not be a valuable addition to the test suite. In the manually analysed subset, this was the only unproductive mutant we identified, highlighting the strength of the *m*-HAZOP and *m*-STPA-generated mutants.

Observation 1: *m*-HAZOP and *m*-STPA can be estimated to have equivalent mutant rates (EMRs) of 3.57% and 16.5% , respectively. Compared to other techniques, *m*-HAZOP had a competitive EMR, whereas *m*-STPA was significantly higher. Unproductive mutants are rare; in the sample of 60 surviving mutants, the authors identified only one example, which future developments in LLM prompting techniques could avoid.

5.5.2 Mutants killed by Defect Triggering Tests

This section will discuss the *m*-HAZOP and *m*-STPA generated mutants that uniquely matched DEFECTS4J bugs in RQ3 (without sampling) and explore the reasons behind it.

Cli-40

In bug 40 of *Cli*, *m*-STPA mutated the `createValue` method in the `TypeHandler` class, to return `null`, instead of throwing a `ParseException` as shown below.

```
-S   throw new ParseException("Unable to handle the class: "
-S                                                  + clazz);
+S   return null;
```

This mutant uniquely mimics the real bug exactly. When describing the method, *m*-STPA identifies the "Throw ParseException" in its control flow description, before

applying the "Does not provide" guide word to produce the UCA of not throwing the exception. It is then implemented by returning `null`, as occurred in the original defect. Such a mutation could be implemented using a rule-based technique; however, MAJOR does not have such an operator and therefore did not produce this mutation. Furthermore, this mutant is larger, with a Levenshtein edit distance of 109, which falls beyond the normal ranges of *m*-LLMORPHEUS and *m*-MUTAHUNTER (see Figure 5.3).

Gson-9

For bug 9 in *Gson*, the `JsonWriter` class' `value` method should perform a null check and return `nullValue()` if the `value` parameter is `null`. In the "buggy" version of this method, there is no null check, resulting in null values being added to an object and `this` being returned, regardless of whether the value is valid or not. The exclusively defect matching (in terms of failing tests) *m*-HAZOP mutant describes this section of code as "Checks if the given value is null" and applies the "Reverse" guide word, creating the deviation "Checks if the given value is not null". As shown below, the mutant then re-implements the behaviour expected later in the method if the value was valid, which includes writing the value as a string to `out` and then returning `this`.

```
1     if (value == null) {
2 +H      out.write("null");
3 -H      return nullValue();
4 +H      return this;
5     }
```

As a result, the mutant behaviour replicates the bug behaviour. The exact mutant syntax would be difficult to mimic using a rule-based tool; however, mutants could be designed to achieve this behaviour. When looking at the behaviours, it appears that the cause is the null check being pseudo-tested (see Chapter 3), and as such, the behaviour could be mimicked using a pseudo-tested statement identification technique (e.g., PseudoSweep (Appendix A)). The LLM-based LLMORPHEUS and MUTAHUNTER approaches seen in this study are also incapable of producing this mutant, as LLMORPHEUS does not target `return` nodes and MUTAHUNTER's prompt set only directs to mutating return types, rather than using returning `this`. As such, it highlights the nuances of the hazard-guided mutation approaches.

Observation 2: *m*-HAZOP and *m*-STPA generated mutants that semantically emulated genuine developer errors that other rule-based and LLM-based techniques could not.

5.5.3 Unique Characteristics of Hazard-Guided LLM-Generated Mutants

In this section, we perform a deeper manual analysis of the 39 remaining mutants from the equivalence analysis (60 mutants minus 20 equivalent and 1 unproductive mutant),

plus an additional 60 killed mutants (i.e., five from each project for each technique), for a total of 99 hazard-guided mutants.

Two authors individually categorised each mutant before meeting to agree on the overall labels and the mutant categorisations. The mutants are divided into the following four categories, as labelled by the authors.

1. **Producible by a rule-based operator (21 mutants)**: These mutants could have been produced using a rule-based Java mutation tool such as MAJOR or PIT.

2. **Producible by a combination of rule-based operators (6 mutants)**: Mutants that could be categorised as "higher-order" mutants, that is, a combination of rule-based mutants, can be used to create these mutants.

3. **Plausibly rule-based (18 mutants)**: Mutants that could be applied given a rule, but are not commonly implemented into widely used tools such as MAJOR or PIT.

4. **Unique to Hazard-Guided approaches (54 mutants)**: Mutants that are tied to the hazard-guided approaches, and are not achievable by the other techniques explored in this study.

The following sections break down the **Plausibly rule-based** and **Unique to Hazard-Guided approaches** mutant categories.

Aggressive versions of rule-based operators

These mutants are **plausibly rule-based** mutants that transform the code using an exaggerated interpretation of an existing rule-based operator. For instance, adding larger constant values, as opposed to traditional value replacements and increments with constants of `1` and `0` (2 mutants); appending string literals (1 mutant) or rewriting a condition that causes an exception (1 mutant). The final condition-rewrite mutant replaced the original type-matching check between the two arrays with a simple length comparison.

Operations that could be rule-based but are not typically implemented

Mutants in this category could, in theory, be implemented as a rule-based operator; however, they are not typically used in state-of-the-art mutation tools. This category included swaps such as one method call for another (of compatible type signature) (1 mutant); `while` keyword for `if` keyword (1 mutant); method parameters for other variables of compatible types (1 mutant) and the invocation target and a compatible parameter from the method call (1 mutant). Furthermore, we also observed mutants reversing the loop iteration direction(2 mutants); deleting `if` statement condition to promote contained statements to the parent scope (7 mutants) and modifying the class used in conjunction with a `synchronize` keyword (1 mutant).

Implementing operators that can achieve these transformations may be difficult in practice, as detailed code analysis, such as type checking, may be necessary to avoid uncompilable mutants.

Aggressive code transformations derived from hazard analysis instructions

Finally, we examine the larger code transformations resulting from the deviations and UCAs generated by the hazard analysis approaches. These mutants included: reordering code statements (13 mutants); substantial rewrite (30 mutants); adding new code (5 mutants) and deleting code (6 mutants). Rule-based techniques often have deletion operators; however, they are usually limited to individual blocks or statements, unlike these hazard-guided mutants. Rule-based approaches also do not add or reorder code statements, which would likely produce several uncompilable mutants. This section's mutants appeared in both the *killed* and *survived* mutants, suggesting that despite the larger code changes, they are not trivial to kill — like the *m*-UNGUIDED mutants — but can aid developers in considering risky behaviours they may not have adequately tested.

Furthermore, the other LLM-based approaches studied in this chapter (i.e., LLMORPHEUS and MUTAHUNTER) are unlikely to generate these large-scale mutants, as they target only AST nodes or individual lines for mutation. Notably, this makes these techniques more comparable to rule-based techniques, such as MAJOR. *m*-HAZOP and *m*-STPA, on the other hand, consider and modify the entire method based on a risky behaviour identified during the hazard analysis process.

Observation 3: The hazard-based LLM-generated techniques produce mutants that existing LLM-based and rule-based techniques do not. Although some mutants could plausibly be applied as rule-based operators, implementing them would be challenging and likely yield many uncompilable mutants. The larger mutation sizes in the hazard analysis techniques are not necessarily trivial, with nuanced behaviours that developer tests did not kill.

5.5.4 The Mutant Utility of Hazard-Based LLM Approaches

This section examines four examples of surviving mutants generated using *m*-HAZOP and *m*-STPA and explores their potential to help a developer improve their test suite.

Chart-1

In the `getLegendItems` method of the `AbstractCategoryItemRenderer`, *m*-HAZOP deletes a null check that should return `result` early, and replaces it with an exception handler that continues method execution even when the value is `null`.

```
1 -H        if (dataset == null) {
2 -H            return result;
3 -H        }
```

```
4  -H        int seriesCount = dataset.getRowCount();
5  +H        int seriesCount;
6  +H        try {
7  +H            seriesCount = dataset.getRowCount();
8  +H        } catch (Exception e) {
9  +H            seriesCount = 0;
10 +H        }
```

This behaviour change is subtle: the mutant may cause the `result` variable to continue being modified before it is returned, whereas the original code returns early to stop this. Notably, the test suite fails to detect this rewrite. This could therefore highlight to the developer that the code may be pseudo-tested or that a subtle behaviour difference has not been considered, both of which may require further testing.

Lang-37

In the `addAll` method in the `ArrayUtils` class, *m*-STPA generated a mutant that rewrote a standard library call to `System.arraycopy` to a `for` loop that implemented the copying process itself.

```
1     final Class<?> type1 = array1.getClass().getComponentType();
2  +S final Class<?> type2 = array2.getClass().getComponentType();
3
4     T[] joinedArray = (T[]) Array.newInstance(type1,
5                                       array1.length + array2.length);
6     System.arraycopy(array1, 0, joinedArray, 0, array1.length);
7  +S int i = 0;
8     try {
9  -S     System.arraycopy(array2, 0, joinedArray,
10 -S                                      array1.length, array2.length);
11 +S     for (; i < array2.length; i++) {
12 +S         joinedArray[array1.length + i] = array2[i];
13 +S     }
14    } catch (ArrayStoreException ase) {
15 -S     final Class<?> type2 = array2.getClass()
16 -S                                     .getComponentType();
17        if (!type1.isAssignableFrom(type2)){
18            throw new IllegalArgumentException(
19                "Cannot store "+type2.getName()
20                +" in an array of "+type1.getName());
21        }
22        throw ase;
23    }
24    return joinedArray;
```

This kind of mutant could be introduced unknowingly by a developer unaware of the `System.arraycopy` library method. While the mutant could be functionally equivalent, it can be caught by paying attention to the expected exceptions from `System.arraycopy` that do not trigger the `IllegalArgumentException` on Line 18, causing the original exception to be re-thrown. While a test suite that does not perform this check may not be a bad test suite, it may leave room for exceptions to be thrown in the wrong areas of the program, causing a test to pass, even if it wasn't the intended exception that was thrown. This

mutant, therefore, has the potential to raise discussion about code implementation and the level of detail in the test suite.

Math-8

This *m*-HAZOP mutant in the `sample` method of the `DiscreteDistribution` class removes the loop iterator for sampling and assignment to the `out` array and fixes sampling to a single sample.

```
-H        for (int i = 0; i < sampleSize; i++) {
-H            out[i] = sample();
+H            out[0] = sample();
-H        }
```

To address this mutant, a developer would need to check more of the state and ensure the number of sampled values in the `out` array was as expected.

Time-6

m-STPA mutated the `getInstance` method in the `GJChronology` class to return the milliseconds of an instant rather than the year. It also reordered the code to perform this check prematurely, before assigning the `cutoverInstance` and `cutoverDate` variables.

```
-S    cutoverInstant = gregorianCutover.toInstant();
-S    LocalDate cutoverDate = new LocalDate(cutoverInstant
-S          .getMillis(), GregorianChronology.getInstance(zone));
-S    if (cutoverDate.getYear() <= 0) {
+S    if (gregorianCutover.toInstant()
+S                        .getMillis() < -62135769600000L) {
              throw new IllegalArgumentException("Cutover too early.
                          Must be on or after 0001-01-01.");
       }
+S    cutoverInstant = gregorianCutover.toInstant();
+S    cutoverDate = new LocalDate(cutoverInstant.getMillis(),
+S                           GregorianChronology.getInstance(zone));
```

The *m*-STPA mutant changes the check from using the local calendar in the chosen timezone to using a raw UTC timestamp. This mutant therefore causes a discrepancy in the first hours of "Year 1" UTC, where the timestamp remains in "Year 0" in other time zones (e.g., New York). This subtle behaviour may therefore be undetected by developer tests.

Observation 4: The *m*-HAZOP and *m*-STPA techniques generate surviving mutants that implement subtle behaviour differences that developer test suites do not check.

5.6 Threats to Validity

Each LLM is trained on a different dataset with different specifications, making each better suited to various tasks. As such, the choice of LLM significantly influences *multiplex*'s

mutant generation, posing a threat to validity. To mitigate this, we employed two distinct models — one locally deployed, smaller model (i.e., gpt-oss:20B) and one larger, cloud-based model (i.e., GPT-4.1). The results, while differing, showed that the techniques under test work with both models and that researchers can replicate them without a cloud-based LLM. That said, both LLM models have a training data cut-off of June 2024, and, as such, they may not reflect the most recent developments. Therefore, future work should further validate the generalisability of these results and *multiplex* by exploring a broader range of models.

As LLMs are inherently non-deterministic, configuration parameters such as temperature and top-p significantly influence their generative output [149]. In alignment with comparable studies (e.g., Wang et al.), we maintained the default temperature settings for both models. Future work should perform a systematic parametric study to explore the impact of these configurations on mutant quality and diversity. Furthermore, using different prompt strategies, including context detail, could influence this study's results; as such, a systematic exploration of variations in prompting strategy is a valuable line of future work for mutation testing with LLMs. Finally, it is possible that *multiplex* contains implementation or LLM interaction defects, or that defects exist in the platform hosting the LLM. We mitigate this by evaluating our tool on small-scale subjects and incorporating a "retry" mechanism for unsuccessful mutant generation, for example, due to rate limits or timeouts. To enable further study of each of these components, the paper this chapter is based on provides a full replication package [132].

As the cut-off date for both LLMs' training data is June 2024, they have likely seen the Defects4J dataset in their training data. Previous studies have mitigated this by using ConDefects; however, as Wu et al. produced this dataset in 2023 (i.e., prior to June 2024), it is likely this also appears in the LLMs' training data [203]. Importantly, neither *m*-HAZOP nor *m*-STPA request the LLM to "implement a defect" but instead guide the LLM through the hazard analysis process, resulting in a pre-specified deviation or unsafe control action being implemented. We see that despite not being asked to implement defects, the hazard analysis approaches both uniquely match defects. However, it is worth noting that *multiplex*'s implementation of other strategies uses their native prompts, which request that it implement defects, so it is plausible that the results for these techniques may be impacted. Notably, these techniques, despite requesting defects, did not consistently match real defects, suggesting the impact may be low.

Because of how they process program control flow, the hazard analysis techniques restrict mutant generation to methods and constructors. While this can exclude the techniques from replicating particular types of defects, they still cover a substantial portion of real-world defects. Future work could, however, investigate how to extend hazard analysis to higher-level control-flow for integration testing and to other Java code elements. Finally, the experiments in this chapter empirically compare the *multiplex* implementa-

tions of HAZOP and STPA with 1) *m*-UNGUIDED, which prompts the LLM to generate a mutant with no additional guidance; 2) two LLM-based techniques LLMORPHEUS and MUTAHUNTER, implemented as *m*-LLMORPHEUS and *m*-MUTAHUNTER respectively and 3) the rule-based MAJOR. In terms of LLM techniques, *m*-UNGUIDED has been chosen as a baseline, while MUTAHUNTER has been chosen as a popular open-source tool from GitHub and LLMORPHEUS as a recent research-based tool. We use MAJOR as a rule-based technique because it integrates with DEFECTS4J, enabling mutant-defect comparison. This empirical evaluation compares hazard analysis techniques with four additional mutation techniques; however, future work should extend this to other LLM- and rule-based techniques.

5.7 Conclusions

This chapter presents a novel approach utilising hazard analysis to guide mutant generation. By leveraging LLM-generated natural-language descriptions (HAZOP) and control-flow abstractions (STPA) of Java methods and constructors, we systematically identify semantic deviations and unsafe control actions to implement as mutants (**C1**). These techniques are presented alongside a comparable suite of LLM-based mutation techniques in *multiplex*, our research-based tool for mutant generation and comparative analysis (**R3**). The results of an empirical evaluation of 15 DEFECTS4J projects demonstrate the following:

1. Larger, more aggressive mutants generated via hazard analysis are significantly more challenging for developer test suites to detect than unguided baselines (**C2**).
2. These results were consistent across both cloud-based and locally deployed LLMs, confirming that hazard analysis techniques can be used on developer workstations (**C1**, **C2**).
3. The hazard-guided mutation techniques effectively complement existing llm- and rule-based techniques by identifying a substantial proportion of uniquely subsuming mutants (**C2**).

The capacity of these techniques to uniquely replicate the behaviour of real-world defects demonstrates their place in advancing mutation testing.

Chapter 6

Conclusions and Future Work

Test suite adequacy is a problem developers face, with little time allotted for testing. However, if their code is not performing as expected, they will face more costly consequences later on. By using unit tests to check the underlying behaviour of their code during development, developers can provide assurance that it behaves as expected.

Code coverage is the most widely used metric to measure test suite adequacy; however, it measures only the quantity of the code executed by the test cases, and does not consider whether an oracle (e.g., an assertion) that checks the behaviour, leaving code vulnerable to pseudo-testedness. Pseudo-testedness can be identified using rule-based deletion mutants; however, such mutants identify only *where* a check is missing and not *why* the check is needed. Mutants require a program context to help developers understand which specific behaviours have not been checked to direct how oracles should be added.

The primary aim of this thesis was to explore and characterise the relationships between mutants, oracles and pseudo-testedness, and to use this characterisation to refine techniques that improve their utility. To address these challenges, I set out to achieve the following high-level objectives in Section 1.4:

1. To **characterise** the relationships between mutants, oracles and pseudo-testedness, specifically through identifying missing assertions or test cases and triggering fault-revealing test cases.

2. To **refine** techniques for producing productive mutants, identifying oracle gaps and evaluating pseudo-testedness.

The following sections summarise my contributions in terms of these objectives.

In Chapter 3, I highlighted the limitations of the existing Extreme mutation testing (XMT) approach for identifying pseudo-tested methods by extending pseudo-testedness evaluation to the statement level. I found that 48% of pseudo-tested statements occur outside of pseudo-tested methods, meaning developers would miss almost half of the pseudo-tested statements in their code if using XMT alone. I also studied the causes of

pseudo-tested statements, demonstrated that they are areas of low fault detection and advanced practitioners' knowledge on how to address them by adding test cases and assertions.

In Chapter 4, I defined the generalised oracle gap to enable comparison of different oracle-based adequacy criteria and their relationship with structural code coverage. I then empirically compared the following three oracle gap calculation approaches (OGCAs): pseudo-tested statement identification (PTSI) (as defined in Chapter 3), checked coverage using a dynamic slice (CC_{DS}) and checked coverage using an observational slice (CC_{OS}). This study found that PTSI identified areas where the tests achieved the lowest traditional mutation score (as calculated using the state-of-the-art PIT mutation tool for Java), compared to CC_{DS} and CC_{OS} and presented the developer with far fewer statements in the fastest execution time. In presenting fewer, but more critical areas to developers, PTSI can enable developers to prioritise the most critical testing deficiencies in their test suite.

In Chapter 5, I demonstrated that applying hazard analysis to LLM-generated code descriptions can lead to semantic implementation differences at the method level, creating mutants that can uniquely improve fault detection. By providing the LLM with guidance from hazard analysis techniques (i.e., HAZOP and STPA) to design mutations, the method-level mutants produced were less trivial and more relevant to the method's context than those from an unguided approach that asked the LLM just to mutate the code. Existing LLM-based and rule-based techniques target small sections of code, such as individual lines or AST nodes, limiting the LLM's mutants to small syntactic changes. As a result, the hazard-guided approach complements existing techniques, generating uniquely subsuming mutants and uniquely identifying defects, enabling developers to identify missing test cases that other methods could not with the same number of mutants.

In conclusion, this thesis refined two techniques for generating productive mutants: pseudo-tested statement identification and applying LLM-guided hazard analysis to mutant generation. It further provided empirical evidence to characterise that these techniques can effectively lead to test cases and assertions that either close an oracle gap or reveal faults. Beyond the immediate practical implications, this thesis has contributed new definitions, empirical evidence and tools to advance knowledge in productive mutant generation and oracle gap calculation. However, no research is without constraints. The following section identifies the limitations of this thesis that prompt future research and development to continue advancing mutant productivity.

6.1 Restrictions and Limitations

The primary limitation of this thesis is that it solely studied Java projects. Particularly in the context of mutation testing, where rule-based mutation operators will likely vary

across tools for different languages, such as StrykerJS for JavaScript and mutmut for Python [81, 4], this creates uncertainty about the broader applicability of these results.

6.1.1 Constraints on Generalisability

This section discusses the constraints on generalisability of this thesis due to the tools and subject sets used and explores the potential impact this has on broader applicability.

Mutation Testing Tools

In Chapter 3, I studied pseudo-testedness in Java projects. However, had this study looked at Python projects, for example, the definitions of statement coverage may yield differing results. We used the PIT mutation tool in Chapter 3 as the baseline, due to its active maintenance and use in related studies. However, using a tool like Major may yield different results. Vice versa, in Chapter 5 we used Major as a rule-based baseline, since the Defects4J integration supports most of the projects under test. However, a tool such as PIT, which uses a different operator set, may yield different results due to the mutants it generates.

Large Language Models

Furthermore, in Chapter 5, we utilise large language models (LLMs) to derive descriptions of existing code, apply hazard analysis and generate mutants. As LLMs are a recent addition to empirical evaluations in mutation testing, best-practice guidelines for comparing LLM-generated mutants remain incomplete, which limits this study. Firstly, the LLM training data includes all available data up to its training cut-off date. Most relevant to this study, this includes the Defects4J project set for both LLMs evaluated. We also only used two models, GPT-4.1 and gpt-oss:20B, for this study. Using a different set of LLMs, including next-generation models, may yield different results (e.g., higher compilability, different mutations, more detailed descriptions) and, as such, lead to different conclusions.

Evaluated Projects

All projects are open-source Java projects either listed on the Maven Central Repository or in the Defects4J benchmark set, with all project code retrieved from GitHub. As a result, these findings may not be representative of results from software written in other programming languages (e.g., Python or JavaScript). Furthermore, these open-source projects may not be representative of industry codebases in terms of testing and development processes, testing requirements and complexity, which may affect how practitioners can apply the findings of this thesis in a proprietary context.

6.1.2 Summary

Overall, the limitations of this thesis stem from tooling and subject selection decisions that enabled a practical evaluation of the research questions. Future work can build on these limitations to understand applicability to mutation testing in practice and other research contexts.

6.2 Future Work

This section discusses plans for future work to build on and extend the work of this thesis.

6.2.1 New Research Questions

Developer Study on the usefulness of pseudo-tested statements in their testing workflow

In Chapters 3 and 4, we studied how pseudo-tested statements could be helpful in practice, showing that they are areas of low traditional mutation score resulting from insufficient assertions or testing. However, these observations did not address how useful pseudo-tested statements are in practice, whether in the open-source community or in industrial contexts. As such, I aim to conduct a developer survey across open source projects to answer the following research questions:

RQ1 How do developers perceive pseudo-tested statements?

RQ2 How does the type of pseudo-tested statement impact how developers address them (e.g., additional testing or code refactoring)?

To explore this in open-source projects, I will use GitHub issue trackers and pull requests to understand developers' responses to these statements and how they contribute to improving their test suites. Furthermore, I intend to conduct an industrial case study to understand whether the approach is applicable in large-team workflows.

Exploring the relationship between oracle gap calculation techniques and "code and test smells"

In Chapters 3 and 4, I explored how different oracle gap calculation techniques performed and the types of statements they found. Notably, code structures, such as data loading and various checks, led to statements appearing in an oracle gap. This raises the question of whether oracle gaps are related to "code smells" and "test smells". Intuitively, if a piece of code is messy, complex and poorly structured, then it is likely also difficult for a developer to write thorough assertions to check it. Similarly, if the tests themselves are

poorly written, it is likely that care has not been taken to ensure they thoroughly check the code. Therefore, to understand the relationships between code and test smells and statements in an oracle gap, I propose answering the following questions:

RQ1 Do specific "code smells" result in larger oracle gaps?

RQ2 Do specific "test smells" result in larger oracle gaps?

Answering these questions provides practical insight into how developers can improve the quality of their code and test suites. If a strong relationship exists between code/test smells and oracle gaps, it suggests a more fundamental issue than isolated missed assertions. If developers can improve their oracle gaps by first addressing code and test smells, it may be more efficient to teach them best practices than to give them another tool to add to their stack.

6.2.2 New Techniques

Automated Pseudo-tested Statement Debugging and Prioritisation

In Chapter 3, we identified the causes of a set of pseudo-tested statements, finding patterns of missing assertions, partial assertions, no targeting test and unintended exception handling. To enable a prioritisation mechanism that presents the most problematic pseudo-tested statements to the developer first, I propose implementing an automated "debugging" process to identify the covering tests and the likely cause of pseudo-testedness, similar to the approach for XMT in Reneri [196]. Prioritising the categories necessitates Section 6.2.1 as a prerequisite to ensure the prioritisations reflect developer objectives. Applying an automated pseudo-tested statement prioritisation would enable a developer to address critical areas first, before moving on to complete mutation testing.

Assertion Generation targeting Pseudo-tested statements

In Chapter 4, we found pseudo-tested statements to be the most effective technique for finding areas of low mutation score. By adding assertion statements to address pseudo-tested code, the developer can likely improve the test suite's mutation score. To continue progressing PTSI, I propose expanding the debugging technique discussed in Section 6.2.2 to inform the automated design and addition of test-suite assertions addressing pseudo-tested statements in the code under test. By introducing an automated technique to add assertions to existing tests, we can reduce the need for developers to spend time adding missing assertions, enabling them to prioritise the bigger issue of missing test cases. After addressing pseudo-tested statements (likely also improving the mutation score), developers can use their limited testing time to target trickier, productive mutants identified by a traditional mutation testing tool. This allows researchers to empirically

investigate how addressing pseudo-tested statements can reduce the number of mutants presented to developers and impact their productivity.

6.2.3 Tooling Improvements

Refine Pseudo-tested Statement Identification

It is important to further study the full impact of pseudo-testedness at the statement level, thus motivating both tool improvements and new experiments. After enhancing PseudoSweep to filter unproductive program elements and better handle test flakiness, we plan to run experiments with additional projects to replicate Chapter 3's experiments, measure the approach's efficiency, and perform studies to understand how PseudoSweep helps software developers. It will also be valuable to implement versions of PseudoSweep in other programming languages to explore how results may differ across different language structures, best practices, and testing infrastructure. Combining Chapter 3's contributions with those arising from future work will position PseudoSweep as a compelling way to detect the pseudo-tested statements neglected by extreme mutation testing, supporting the identification of testing inadequacies that developers can address before committing resources to traditional mutation testing.

Integrating oracle gap calculation into developer workflows

In Chapter 4, we studied how different oracle gaps can reveal weaknesses in test suite fault detection. However, setup and execution times are a barrier to developer adoption of such techniques. By integrating the oracle gap calculation into continuous integration (CI) pipelines, such as GitHub Actions, we can enable developers to use it as a testing criterion when pushing code.

6.2.4 Benchmarks

Publish an LLM-based Mutation Analysis Benchmark

In Chapter 5, we investigated the role of hazard analysis in guiding LLMs to generate method-level mutations. A problem we faced at the time was identifying best practices for LLM mutant comparison, as standard rule-based comparison techniques do not account for the costs and processes involved in generating LLM mutants. I therefore plan to review existing studies on LLMs in mutation testing and to collate best-practice guidelines for the empirical evaluation of LLM-based mutation techniques. This will enable a publicly ranked LLM-based Mutation Analysis benchmark using predetermined metrics, for researcher and practitioner use and contribution.

6.2.5 Summary

This section outlines a roadmap for extending the work presented in this thesis to advance knowledge and practical application of pseudo-tested statements, oracle gaps, and LLMs in mutant generation. By implementing and exploring each of these areas, future work can provide developers with actionable insights and improve their test suites.

Bibliography

[1] Demo replication: `https://github.com/PseudoTested/pseudosweep-demo`.

[2] Maven central repository: `https://repo.maven.apache.org/maven2`.

[3] Pseudosweep tool: `https://github.com/PseudoTested/PseudoSweep`.

[4] StrykerJS (`https://github.com/stryker-mutator/stryker-js`).

[5] Where tests fall short - replication package (`https://github.com/pseudotested/esem-2025-replication-package`).

[6] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[7] JaCoCo: Java code coverage library (`https://www.jacoco.org/jacoco/`), 2014.

[8] IEC 61882:2016 Hazard and operability studies (HAZOP studies) – Application guide, 2016.

[9] JavaSlicer (https://github.com/backes/javaslicer), 2016 (Last Updated).

[10] Pit mutators (https://pitest.org/quickstart/mutators/), 2025.

[11] K. Ahmed, M. Lis, and J. Rubin. Slicer4j: a dynamic slicer for java. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1570—-1574, 2021.

[12] Z. Ahmed, E. Schwass, S. Herbold, F. Trautsch, and J. Grabowski. A new perspective on the competent programmer hypothesis through the reproduction of real faults with repeated mutations. *Software Testing, Verification and Reliability*, 34(3), 2024.

[13] P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of International Conference on Software Testing, Verification and Validation*, 2014.

[14] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2016.

[15] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering*, 2005.

[16] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[17] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[18] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[19] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professiona, 2002.

[20] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.

[21] M. Beller, C-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 268–277, 2021.

[22] M. Betka and S. Wagner. Extreme mutation testing in practice: An industrial case study. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 113–116, 2021.

[23] M. Betka and S. Wagner. Towards practical application of mutation testing in industry — Traditional versus extreme mutation testing. *Journal of Software: Evolution and Process*, 34(11), 2022.

[24] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-Independent Program Slicing. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 109—-120, 2014.

[25] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS and the Limits of Static Slicing. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–10, 2015.

[26] M. S.h Bouafif, M. Hamdaqa, and E. Zulkoski. Primg: Efficient llm-driven test generation using mutant prioritization. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, pages 1107—-1116, 2025.

[27] D. Bowes, T. Hall, J. Petrić, T. Shippey, and B. Turhan. How good are my tests? In *Proceedings of the Workshop on Emerging Trends in Software Metrics*, 2017.

[28] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps. The Care and Feeding of Wild-Caught Mutants. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 511—522, 2017.

[29] T.A Budd and D. Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18:31–45, 1982.

[30] M. A. Cachia, M. Micallef, and C. Colombo. Towards Incremental Mutation Testing. *Electronic Notes in Theoretical Computer Science*, 294:2–11, 2013.

[31] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen. Coding In-depth Semistructured Interviews: Problems of Unitization and Intercoder Reliability and Agreement. *Sociological Methods & Research*, 42(3), 2013.

[32] S. Charalampidou, A. Zeleskidis, and I. M. Dokas. Hazard analysis in the era of ai: Assessing the usefulness of chatgpt4 in stpa hazard analysis. *Safety Science*, 178, 2024.

[33] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[34] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota. An empirical study on the usage of transformer models for code completion. *Transactions on Software Engineering*, 48(12), 4818–4837.

[35] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 196—206, 2007.

[36] CodeIntegrity. Mutahunter (https://github.com/codeintegrity-ai/mutahunter), 2025.

[37] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A practical mutation testing tool for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2016.

[38] C. Comar, J. Guitton, O. Hainque, and T. Quinot. Formalization and Comparison of MCDC and Object Branch Coverage Criteria. In *Proceedings of Embedded Real Time Software and Systems (ERTS)*, 2012.

[39] O.J. Dahl, E.W. Dijkstra, E.W. Dijkstra, and C.A.R Hoare. *Structured Programming*. Academic Press, 1972.

[40] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology*, 171, 2024.

[41] R. Degiovanni and M. Papadakis. µBert: Mutation Testing using Pre-Trained Language Models. In *Proceedings of International Conference on Software Testing, Verification and Validation Workshops*, 2022.

[42] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert. Code Generation Using Machine Learning: A Systematic Review. *IEEE Access*, 10, 2022.

[43] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt. Experimental evaluation of SDL and one-op mutation for C. In *Proceedings of International Conference on Software Testing, Verification and Validation*, 2014.

[44] M. E. Delamaro, J. Offutt, and P. Ammann. Designing deletion mutation operators. In *Proceedings of International Conference on Software Testing, Verification and Validation*, 2014.

[45] P Delgado-Pérez, I. Medina-Bulo, Palomo-Lozano. F., A. García-Domínguez, and J. J. Domínguez-Jiménez. Assessment of class mutation operators for C++ with the mucpp mutation system. *Information & Software Technology*, 81:169–184, 2017.

[46] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. Clark, and I. Medina-Bulo. Evaluation of mutation testing in a nuclear industry case study. *IEEE Transactions on Reliability*, 67(4):1406–1419, 2018.

[47] R.A. DeMillo and Offutt J. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9), 1991.

[48] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), 1978.

[49] L. Deng, J. Offutt, and N Li. Empirical evaluation of the statement deletion mutation operator. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2013.

[50] A. Derezińska. Evaluation of Deletion Mutation Operators in Mutation [t]esting of C# Programs. In *Proceedings of the International Conference on Dependability and Complex Systems*, pages 97–108, 2016.

[51] A. Derezińska and K. Hałas. Analysis of mutation operators for the python language. In *Proceedings of the International Conference on Dependability and Complex Systems*, pages 155–164, 2014.

[52] S. Diemert and J. H. Weber. Can Large Language Models Assist in Hazard Analysis? In *Computer Safety, Reliability, and Security. SAFECOMP 2023 Workshops: ASSURE, DECSoS, SASSUR, SENSEI, SRToITS, and WAISE, Toulouse, France, September 19, 2023, Proceedings*, pages 410—422, 2023.

[53] J Dunjó, V. Fthenakis, J. A. Vílchez, and J. Arnaldos. Hazard and operability (hazop) analysis. a literature review. *Journal of Hazardous Materials*, 173(1), 2010.

[54] V. H. S. Durelli, N. M. De Souza, and M. E. Delamaro. Are deletion mutants easier to identify manually? In *Proceedings of International Workshop on Mutation Testing*, 2017.

[55] C. A. Ericson. *Hazard Analysis Techniques for System Safety*. John Wiley & Sons, 2nd edition, 2015.

[56] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, Santos A. L. M., A Cavalcanti, F. C. Ferrari, and J C Maldonado. Avoiding useless mutants. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, 2017.

[57] T. Fidry. Humbug: Mutation testing for PHP (`https://github.com/humbug/humbug`), 2017.

[58] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10), 1988.

[59] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416—419, 2011.

[60] G. Fraser and A. Arcuri. A Retrospective on Whole Test Suite Generation: On the Role of SBST in the Age of LLMs. *IEEE Transactions on Software Engineering*, 51(3):874–878, 2025.

[61] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3), 1993.

[62] A. Garg, R. Degiovanni, M. Papadakis, and Y. Le Traon. On the Coupling between Vulnerabilities and LLM-Generated Mutants: A Study on Vul4J Dataset . In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 305–316, 2024.

[63] G. Gay and A. Salahirad. How closely are common mutation operators coupled to real faults? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 129–140, 2023.

[64] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl. Automated oracle data selection support. *IEEE Transactions on Software Engineering*, 41(11), 2015.

[65] N. E. Gold, D. Binkley, M. Harman, S. Islam, J. Krinke, and S. Yoo. Generalized observational slicing for tree-represented modelling languages. In *Proceedings of the Foundations of Software Engineering*, pages 547–558, 2017.

[66] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *SIGPLAN Notices*, 10(6), 1975.

[67] R. Gopinath, I. Ahmed, M-A. Alipour, C. Jensen, and A. Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 66(3), 2017.

[68] R. Gopinath, C. Jensen, and A. Groce. Code Coverage for Suite Evaluation by Developers. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 72—82, 2014.

[69] R. Gopinath, B. Mathis, and A. Zeller. If you can't kill a supermutant, you have a problem. In *International Workshop on Mutation Testing*, 2018.

[70] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser. Do automatic test generation tools generate flaky tests? In *Proceedings of the International Conference on Software Engineering*, 2024.

[71] H. G. Gurbuz, B. Tekinerdogan, C. Catal, and N. P. Er. Test suite assessment of safety-critical systems using safety tactics and fault-based mutation testing. *Cluster Computing*, 27(4), 2024.

[72] A. Hamidi, A. Khanfir, and M. Papadakis. Intent-Based Mutation Testing: From Naturally Written Programming Intents to Mutants. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 347–357, 2025.

[73] M. Harman and R. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[74] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo. Angels and monsters: an empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 397—408, 2014.

[75] M. Harman, J. Ritchey, I. Harper, S. Sengupta, K. Mao, A. Gulati, C. Foster, and H. Robert. Mutation-guided LLM-based test generation at Meta. In *Proceedings of the International Conference on the Foundations of Software Engineering*, pages 180–191, 2025.

[76] H. Hemmati. How effective are code coverage criteria? In *Proceedings of the International Conference on Software Quality, Reliability and Security*, 2015.

[77] R. Hierons, M. Harman, and S. Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification and Reliability*, 9(4), 1999.

[78] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 197–207, 2017.

[79] S. B. Hossain and M. B. Dwyer. A brief survey on oracle-based test adequacy metrics. In *arXiv:2212.06118*, 2019.

[80] S. B. Hossain, M. B. Dwyer, S. Elbaum, and A. Nguyen-Tuong. Measuring and mitigating gaps in structural testing. In *Proceedings of the International Conference on Software Engineering*, 2023.

[81] A. Hovmöller. mutmut - python mutation tester (`https://github.com/boxed/mutmut`), 2026.

[82] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. DeepMutation++: A mutation testing framework for deep learning systems. In *Proceedings of the International Conference on Automated Software Engineering*, 2019.

[83] C. Huo and J. Clause. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 621—-631, 2014.

[84] C. Huo and J. Clause. Interpreting coverage information using direct and indirect coverage. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2016.

[85] A. R. Ibrahimzada, Y. Chen, R. Rong, and R. Jabbarvand. Challenging bug prediction and repair models with synthetic bugs. In *Proceedings of the International Conference on Source Code Analysis and Manipulation*, 2025.

[86] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, pages 435—-445, 2014.

[87] S. Islam and D. Binkley. Porbs: A parallel observation-based slicer. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 1–3, 2016.

[88] M. Ivanković, G. Petrović, R. Just, and G. Fraser. Code coverage at google. In *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955—963, 2019.

[89] M. Ivankovic, G. Petrovic, Y. Kulizhskaya, M. Lewko, L. Kalinovcic, R. Just, and G. Fraser. Productive Coverage: Improving the Actionability of Code Coverage. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, pages 58—-68, 2024.

[90] G. Jahangirova, D. Clark, M. Harman, and P. Tonella. Test Oracle Assessment and Improvement. In *Proceeedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 247–258, 2016.

[91] G. Jahangirova, D. Clark, M. Harman, and P. Tonella. OASIs: Oracle Assessment and Improvement Tool. In *Proceesings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 368–371, 2018.

[92] G. Jahangirova, D. Clark, M. Harman, and P. Tonella. An Empirical Validation of Oracle Improvement. *IEEE Transactiions on Software Engineering*, 47(8), 2021.

[93] Y. Jia and M. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, 2008.

[94] Y. Jia and M. Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10), 2009.

[95] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2011.

[96] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of International Symposium on Software Testing and Analysis*, 2014.

[97] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2014.

[98] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014.

[99] R. Just, G. M. Kapfhammer, and F. Schweiggert. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering*, 2011.

[100] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the International Workshop on Automation of Software Test*, 2011.

[101] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering*, 2012.

[102] G. K. Kaya, D Bovell, M. Sujan, and G. Braithwaite. Large language models powered system safety assessment: applying stpa and fram. *Safety Science*, 191, 2025.

[103] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. Le Traon. Efficient mutation testing via pre-trained language models. *arXiv preprint arXiv:2301.03543*, 2023.

[104] M. Kim, N. Kim, and H. P. In. Investigating the Relationship Between Mutants and Real Faults with Respect to Mutated Code. *International Journal of Software Engineering and Knowledge Engineering*, 30(08):1119–1137, 2020.

[105] S. Kim, J. A. Clark, and J. A. McDermid. The rigorous generation of Java mutation operators using HAZOP. Technical report 2/8/99, Department of Computer Science, University of York, York, UK, 1999.

[106] M. Kintis and N. Malevris. MEDIC: A static analysis framework for equivalent mutant identification. *Information and Software Technology*, 68:1–17, 2015.

[107] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. Detecting Trivial Mutant Equivalences via Compiler Optimisation . *IEEE Transactions on Software Engineering*, 44(04), 2018.

[108] T. A. Kletz. *HAZOP and HAZAN: Identifying and Assessing Process Industry Hazards*. Institution of Chemical Engineers, Rugby, UK, 4th edition, 1999.

[109] P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, 2015.

[110] R. Koitz-Hristov, L. Stracke, and F. Wotawa. Checked coverage for test suite reduction — is it worth the effort? In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 6–16, 2022.

[111] B. Korel and J. Laski. DYNAMIC PROGRAM SLICING. *Information Processing Letters*, 29(3), 1988.

[112] K. Koster. A state coverage tool for JUnit. In *Proceedings of the Companion of the International Conference on Software Engineering*, 2008.

[113] K. Koster and D. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering*, 2007.

[114] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng. Mutant subsumption graphs. In *Proceedings of the Mutation Analysis Workshop, International Conference on Software Testing, Verification, and Validation Workshops*, 2014.

[115] B. Kushigian, A. Rawat, and R. Just. Medusa: Mutant equivalence detection using satisfiability analysis. In *Proceedings of the International Workshop on Mutation Analysis*, 2019.

[116] J. D. Lawrence and J. M. Gallagher. A proposal for performing software safety hazard analysis. *Reliability Engineering & System Safety*, 55(3), 1997.

[117] J. Lee, S. Park, S. Oh, and B Ma. Can large language models automate the hazop process without human intervention? *Safety Science*, 194:107039, 2026.

[118] N. G. Leveson. A new accident model for engineering safer systems. *Safety Science*, 42(4), 2004.

[119] N. G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety.* MIT Press, 2011.

[120] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu. TransRepair: Context-aware Program Repair for Compilation Errors. In *Proceedings of the International Conference on Automated Software Engineering*, pages 1368–1380, 2022.

[121] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel. Dodona: automated oracle data set selection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 193–203, 2014.

[122] S. Lukasczyk and G. Fraser. Pynguin: automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168—172, 2022.

[123] Y-S. Ma and J. Offutt. Description of muJava's method-level mutation operators. Technical report, Electronics and Telecommunications Research Institute, Korea, 2005, updated 2016.

[124] Y-S. Ma, J. Offutt, and Y-R. Kwon. muJava: An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2), 2005.

[125] Y-S. Ma, J. Offutt, and Y-R. Kwon. muJava: A mutation system for Java. In *Proceedings of the International Conference on Software Engineering*, 2006.

[126] L. Madeyski, W. Orzeszyna, R. Torkar, and M Józala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering*, 40(1), 2014.

[127] A. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[128] M. Maton. Exploring pseudo-testedness - replication package(`https://github.com/PseudoTested/icsme-2024-replication-package`), 2024.

[129] M. Maton, G. M. Kapfhammer, and P. McMinn. Exploring Pseudo-Testedness: Empirically Evaluating Extreme Mutation Testing at the Statement Level. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2024.

[130] M. Maton, G. M. Kapfhammer, and P. McMinn. PseudoSweep: A pseudo-tested code identifier. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), Tool Track*, 2024.

[131] M. Maton, G. M. Kapfhammer, and P. McMinn. Where Tests Fall Short: Empirically Analyzing Oracle Gaps in Covered Code. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2025.

[132] M. Maton, G. M. Kapfhammer, and P. McMinn. Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2026.

[133] Y. Meng, G. Gay, and M. Whalen. Ensuring the observability of structural test obligations. *Transactions on Software Engineering*, 46(7), 2020.

[134] J.C. Miller and C.J. Maloney. Systematic Mistake Analysis of Digital Computer Programs. *Communications of the ACM*, 6(2), 1963.

[135] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing (Third Edition)*. John Wiley & Sons, Inc., 2011.

[136] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, volume 5, 2005.

[137] R. Niedermayr, E. Jürgen, and S. Wagner. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pages 23–29, 2016.

[138] A. J. Offutt. The coupling effect: fact or fiction. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification*, pages 131—-140, 1989.

[139] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1), 1992.

[140] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3), 1994.

[141] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3), 1997.

[142] A. J. Offutt and R. H. Untch. *Mutation Testing for the New Century*, chapter Mutation 2000: Uniting the Orthogonal. Springer, 2001.

[143] A.J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the International Conference on Software Engineering*, pages 100–107, 1993.

[144] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis, and Y. Le Traon. Syntactic Versus Semantic Similarity of Artificial and Real Faults in Mutation Testing Studies. *IEEE Transactions on Software Engineering*, 49(7), 2023.

[145] C. Olszowka. SimpleCov (https://github.com/simplecov-ruby/simplecov).

[146] OpenAI. Introducing GPT-OSS. *https://openai.com/index/introducing-gpt-oss/*, 2021.

[147] OpenAI. gpt-oss-20b model card, 2025.

[148] OSSF. Criticality score v2.0.4 (https://openssf.org/projects/criticality-score/), 2024.

[149] S. Ouyang, J. M. Zhang, M. Harman, and Meng Wang. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2), 2025.

[150] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the International Conference on Software Engineering*, pages 936—-946, 2015.

[151] M. Papadakis, M. Kintis, J. Zhang, Y Jia, Y. Le Traon, and M. Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–0378. 2019.

[152] M. Papadakis and N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*, pages 90–99, 2010.

[153] M. Papadakis, D. Shin, S. Yoo, and D-H. Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*, pages 537—-548, 2018.

[154] O. Parry, M. Hilton, G. M. Kapfhammer, and P. McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology*, 31(1), 2022.

[155] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. What do developer-repaired flaky tests tell us about the effectiveness of automated flaky test detection? In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 160–164, 2022.

[156] H. Pasman and W. Rogers. How Can We Improve HAZOP, Our Old Work Horse, and Do More with Its Results? An Overview of Recent Developments. *Chemical Engineering Transactions*, 48:829–834, 2016.

[157] J. Patra and M. Pradel. Semantic bug seeding: A learning-based approach for creating realistic bugs. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 906—918, 2021.

[158] M. Patrick, M. Oriol, and J. A. Clark. MESSI: Mutant Evaluation by Static Semantic Interpretation. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 711–719, 2012.

[159] G. Petrovic and M. Ivankovic. State of mutation testing at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2018.

[160] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just. Does Mutation Testing Improve Testing Practices? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 910–921, 2021.

[161] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just. Practical mutation testing at scale: A view from Google. *IEEE Transactions on Software Engineering*, 48(10), 2022.

[162] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just. Please fix this mutant: How do developers resolve mutants surfaced during code review? In *Proceedings of the International Conference on Software Engineering*, 2023.

[163] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, 2018.

[164] P. Piwowarski, Mi. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of the International Conference on Software Engineering*, pages 287—301, 1993.

[165] A. V Pizzoleto, F. C Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157, 2019.

[166] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, 1985.

[167] M. Rausand. *Risk Assessment; Theory, Methods, and Applications*. Wiley, 2011.

[168] C. Richter and H. Wehrheim. Learning realistic mutations: Bug creation for neural bug detectors. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 162–173, 2022.

[169] S. Rismani, R. Shelby, A. Smart, R. Delos Santos, A. Moon, and N. Rostamzadeh. Beyond the ML Model: Applying Safety Engineering Frameworks to Text-to-Image

Development. In *In Proceedings of the Conference on AI, Ethics and Society*, pages 70–83, 2023.

[170] A. B Sánchez, P. Delgado-Pérez, I Medina-Bulo, and S. Segura. Mutation Testing in the Wild: Findings from GitHub. *Empirical Software Engineering*, 27(6), 2022.

[171] A. B. Sánchez, J. A. Parejo, S. Segura, A. Durán, and M. Papadakis. Mutation testing in practice: Insights from open-source software developers. *IEEE Transactiions on Software Engineering*, 50(5), 2024.

[172] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *Proceedings of the International Conference on Automated Software Engineering*, pages 343—-352, 2007.

[173] M. Schirp. mutant (`https://github.com/mbj/mutant`), 2026.

[174] D Schuler and A. Zeller. Assessing Oracle Quality with Checked Coverage. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.

[175] D. Schuler and A. Zeller. Checked coverage: An indicator for oracle quality. *Software Testing, Verification and Reliability*, 23(7), 2013.

[176] A. Shi, J. Bell, and D. Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2019.

[177] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, pages 351—-360, 2008.

[178] N. Smith, D. van Bruggen, and F. Tomassetti. *JavaParser: Visited*. LeanPub, 2023.

[179] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 391—400, 2011.

[180] P. Straubinger and G. Fraser. A survey on what developers think about testing. In *Proceedings of the 34th International Symposium on Software Reliability Engineering*, 2023.

[181] P. Straubinger, M. Kreis, S. Lukasczyk, and G. Fraser. Mutation Testing via Iterative Large Language Model-Driven Scientific debugging. In *Proceedings of International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 358–367, 2025.

[182] J. P. Thomas. Investigation of the use of system-theoretic process analysis at the NRC. Technical Report ML22272A315, U.S. Nuclear Regulatory Commission, 2021.

[183] J. P. Thomas and J. G. Van Houdt. Evaluation of system-theoretic process analysis (STPA) for improving aviation safety. Technical Report DOTFAATC-2416, Federal Aviation Administration, William J. Hughes Technical Center, 2024.

[184] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang. Learning to construct better mutation faults. In *Proceedings of the International Conference on Automated Software Engineering*, 2023.

[185] Z. Tian, H. Shu, D. Wang, X. Cao, Y. Kamei, and J. Chen. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1733–1745, 2024.

[186] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[187] F. Tip, J. Bell, and M. Schaefer. LLMorpheus: Mutation testing using large language models. *IEEE Transactions on Software Engineering*, 51, 2025.

[188] M. Tufano, J. Kimko, S. Wang, C. Watson, G. Bavota, M. Di Penta, and D. Poshyvanyk. DeepMutation: A neural mutation tool. In *Proceedings of the International Conference on Software Engineering: Companion Proceedings*, 2020.

[189] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. Learning how to mutate source code from bug-fixes. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312, 2019.

[190] R.H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Proceedings of the Annual Southeast Regional Conference*, 2009.

[191] R.H. Untch, A.J. Offutt, and M.J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of International Symposium on Software Testing and Analysis*, 1993.

[192] A. van Deurson, L. Moonen, A. van den Bergh, and G. Kok. Refactoring Test Code. Technical report, CWI (Centre for Mathematics and Computer Science), 2001.

[193] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens. State coverage: Software validation metrics beyond code coverage. In *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science*, 2012.

[194] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000–6010, 2017.

[195] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, 24(3), 2019.

[196] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. Suggestions on test suite improvements with automatic infection and propagation analysis. In *arXiv:1909.04770*, 2019.

[197] O. L. Vera-Pérez, M. Monperrus, and B. Baudry. Descartes: A PITest engine to detect pseudo-tested methods. In *Proceedings of the International Conference on Automated Software Engineering*, 2018.

[198] B. Wang, M. Chen, M. Deng, Y. Lin, M. Harman, M. Papadakis, and J. M. Zhang. A comprehensive study on large language models for mutation testing. *ACM Transactions on Software Engineering and Methodology*, 2026.

[199] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7), 1982.

[200] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[201] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 102–111, 2013.

[202] C.-P. Wong, J. Meinicke, L. Chen, J. P. Diniz, C. Kästner, and E. Figueiredo. Efficiently finding higher-order mutants. In *Proceedings of the Joint Meeting on European Software Engineering Conference and International Symposium on the Foundations of Software Engineering*, 2020.

[203] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu. Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024.

[204] Qian Yang, J. Jenny Li, and David M. Weiss. A survey of coverage-based testing tools. *Computer Journal*, 52(5):589–597, 2009.

[205] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the International Conference on Software Engineering*, pages 919—930, 2014.

[206] H. Zhang, T. Yue, S. Ali, and C Liu. Towards mutation analysis for use cases. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, 2016.

[207] J. M. Zhang, L. Zhang, D. Hao, M. Wang, and L. Zhang. Do Pseudo Test Suites Lead to Inflated Correlation in Measuring Test Effectiveness? In *Proceedings of International Conference on Software Testing, Validation and Verification (ICST)*, pages 252–263, 2019.

[208] Y. Zhang and A. Mesbah. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proceedings of Joint Meeting on Foundations of Software Engineering*, pages 214–224, 2015.

[209] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Compututing Surveys*, 29(4), 1997.

[210] Q. Zhu, A. Zaidman, and A. Panichella. How to kill them all: An exploratory study on the impact of code observability on mutation testing. *Journal of Systems and Software*, 173, 2021.

Appendix A

PseudoSweep: A Pseudo-Tested Code Identifier

The contents of this appendix are based on the paper "M. Maton, G. M. Kapfhammer, and P. McMinn. PseudoSweep: A pseudo-tested code identifier. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), Tool Track*, 2024" (© 2024 IEEE).

Author Contribution For this publication, I was responsible for extending and repurposing a prototype created by the third author to integrate and extend statement deletion to detect pseudo-tested statements, under the guidance and supervision of the second and third authors. I drafted the paper, incorporating the second and third authors' feedback and text edits for the final version.

Use in this Chapter For this chapter, the content of the original paper has been adapted and edited to align with the background provided in Chapter 2. To maintain a consistent voice throughout this thesis, I have independently re-authored all sections drafted by co-authors.

A.1 Introduction

If a tool can remove production code without causing the test suite to fail, it is difficult to argue that the code is thoroughly tested. This scenario, in which a test executes some program code, and yet a tool can remove it without causing a test to fail, means that the code is "pseudo-tested", which is a quality risk [137].

For instance, Listing A.1 shows a *formatTag* method and highlights its pseudo-tested statement on Line 3. The *testFormatTag* test executes all lines in this method but only contains assertions that check the start and end tags. This means the statement on Line 3, which places the content between the tags, has no observable impact on the

method's output. Even though the test executes this statement, a developer can delete it without causing any tests to fail. Such code is either redundant or requires further testing to detect its removal correctly.

Revealing pseudo-tested code at early testing stages will enable developers to identify weaknesses in their test suite before progressing to more expensive approaches. For example, traditional mutation testing tools evaluate a test suite's bug-finding capability by inserting synthetic defects into the code and executing the test suite against them to see if they cause tests to fail [15, 48]. Despite extensive research and development, mutation testing often remains costly [165].

Developers can identify pseudo-tested methods using a branch of mutation testing called extreme mutation testing (XMT), which removes entire method bodies (using a dummy return value when required for compilation) and executes the tests [137]. Prior work has found pseudo-tested methods in well-tested subjects, including open-source Apache projects [137, 195]. However, as found in Chapter 3, non-pseudo-tested methods still contained pseudo-tested statements [129]. Highlighting such statements will help developers address simple testing issues such as missing tests and partial assertions.

This appendix introduces PseudoSweep, a tool that highlights pseudo-tested statements and methods in Java projects (supporting JDK versions 8–12), thus enabling developers to address testing inadequacies. Ensuring that the program always compiles, PseudoSweep systematically removes each covered method and statement from the program under test's source code and runs the test suite to detect elements it can remove without affecting test outcomes. It reveals these elements to developers so that they can identify the testing deficiency that may have caused it. This tool provides a novel implementation for identifying pseudo-tested statements, helping developers address testing inadequacies before committing resources to traditional mutation testing.

To demonstrate how developers can use our tool, we ran it on the *Caverphone2* class from *commons-codec* as discussed in Chapter 3. We walk through the outputs for the *encode* method and contrast the findings from running PseudoSweep with what code coverage would have revealed. We also provide an overview of experimental results from Chapter 3 exploring the frequency and causes of pseudo-tested statements in 27 real-world Java projects in the context of the implementation.

A.2 PseudoSweep: Detecting Pseudo-tested Elements in Project Source Code

We designed PseudoSweep to identify and report pseudo-tested statements and methods within the source code of a Java project. By revealing these testing deficiencies before developers commit resources to traditional mutation testing, PseudoSweep pro-

```
1  public String formatTag(String tag, String content) {
2      String html = "<" + tag + ">";
3      html += content;
4      html += "</" + tag + ">";
5      return html;
6  }
7
8  @Test
9  public void testFormatTag() {
10     String str = formatTag("p", "hello world!");
11     assertThat(str, startsWith("<p"));
12     assertThat(str, endsWith("/p>"));
13 }
```

Listing A.1: The source code of the *formatTag* method and its corresponding test case called *testFormatTag*. The PSEUDOSWEEP tool presented in this chapter can automatically determine that the statement on line 3 is pseudo-tested (© 2024 IEEE).

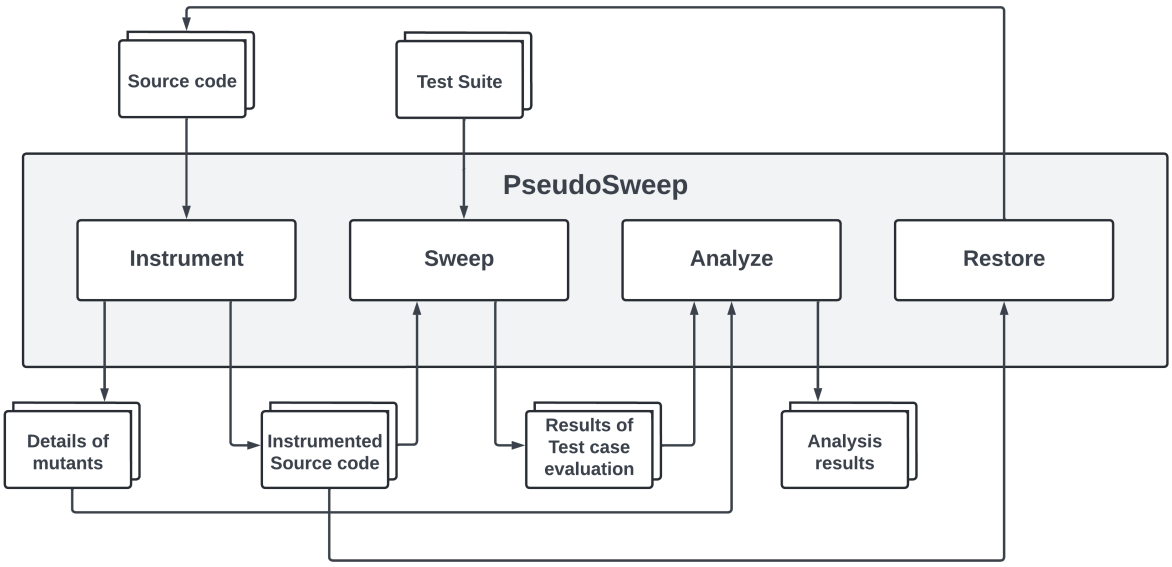

Figure A.1: Diagram of PSEUDOSWEEP's main components. Instrument, Sweep, Analyze and Restore are entry points, run from the command line (© 2024 IEEE).

vides helpful insights relevant to the test suite's maturity. The tool focuses on detecting pseudo-tested elements through deletion mutations. To achieve this, we designed and implemented a set of appropriate operators. Figure A.1 overviews the core components of PSEUDOSWEEP.

A.2.1 Instrument

To keep the program's source code, PSEUDOSWEEP duplicates each Java Class file to a temporary one that has the *.java.orig* file extension, enabling the instrumented source code to be reverted back to the original with the Restore function.

Table A.1: Descriptions of the deletion mutation operators employed by PSEUDOSWEEP. In the "Example Transformation" column, the method calls to `exec`, `fix` and `eval` are simplified. In the full instrumentation inserted by the tool, this source code would also include class information and the element type (© 2024 IEEE).

Operator: Description	**Applicable Element Types**	**Example Source Code**	**Example Transformation**

MDR: Delete method body and return default value	Non-void Method	`public int method () {`
		`  methodBody;`
		`  return 5;`
		`}`
			`public int method () {`
			`  if (exec(id)) {`
			`  methodBody;`
			`  return 5;`
			`  }`
			`  return (execDefault(id)) ? 0 : 1;`
			`}`

MDV: Delete method body	Void Method	`public void method () {`
		`  methodBody;`
		`}`
			`public void method () {`
			`  if (exec(id)) {`
			`  methodBody;`
			`  }`
			`}`

SDSS: Delete Statement	Break, Continue, Do-while, Expression, For, For Each, If, Contains Inner Class, Lambda, Switch, Try, While	`statement;`	`if (exec(id)) {`
			`  statement;`
			`}`

SDSL: Delete statement with label	Break, Continue	`statement label;`	`if (exec(id)) {`
			`  statement label;`
			`}`

SDSR: Delete the statement and return default value	Return	`return "string";`	`if (exec(id)) {`
			`  return "string";`
			`}`
			`return (execDefault(id)) ? "" : "A";`

SDVD: Delete declaration initialisation	Variable declarations	`int i=3;`	`int i=0;`
			`if (exec(id)) {`
			`  i=3;`
			`}`

SFCT: Fix conditional expression to True	Do-while, For, If, While	`if ( a < b ) {` `}`	`if(fix(eval() && a < b, id)){` `}`

SDLL: Delete loop statement with label	For, While	`label:`
		`for(){`
		`}`
			`if (exec(id)) {`
			`  label:`
			`  for(){`
			`  }`
			`}`

Operators

Detecting pseudo-tested elements requires operators that can delete code elements. The current version of the tool operates at the statement and method levels; therefore, the operator-set design targets these component types. We look to extend this in future tool iterations to give users control over the granularity of the results they receive.

Method Operators We have developed PSEUDOSWEEP's method-level operators based on the DESCARTES Mutation Engine for PIT set and described them in Table A.1 [197]. If a Java method declares a non-void return type, then the method requires at least one *return* statement at the end of the method to compile. To check if a method is pseudo-tested, PSEUDOSWEEP must delete the whole method. Therefore, we must add a placeholder return statement to ensure compilation. We use two default return values

Table A.2: Default return values for return statements (© 2024 IEEE).

Type	**Default Values**	
Boolean	false	true
Byte	0	1
Double	0	1
Float	0	1
Char	‘ ’ (space character)	‘A’
Short	0	1
Int	0	1
String	“”	“A”
Long	0	1
Collection	null	New ArrayList
Iterable	null	New ArrayList
List	null	New ArrayList
Queue	null	New LinkedList
Set	null	New HashSet
Map	null	New HashMap
Reference Type	null	null

for each return statement to mitigate the risk of returning the expected value. Table A.2 presents the default method return values.

Statement Operators We have assembled six operators to detect pseudo-tested statements, as described in Table A.1. We looked to the statement deletion operator (SDL) defined by Deng et al. [49]; however, their operator set required additional operators, such as a *return value mutation operator*, to be suitable for identifying pseudo-tested statements. For example, SDL provides a single default return value for all types except Boolean. These single defaults leave room for equivalent mutants, such as returning “0” when an assertion expects “0”. To overcome this problem, the DESCARTES Mutation Engine for PIT applies two default-value mutants and adds default values for additional types, such as Collections and Maps. We have extended PSEUDOSWEEP’s implementation of SDL to include this as presented in Table A.2.

Metamutant

PSEUDOSWEEP implements these operators by creating a metamutant [191] that adds *if* statements to the source code of the project that PSEUDOSWEEP manipulates to “remove” individual code elements during a “Sweep” (Section A.2.2). The metamutant approach ensures that mutants always map directly to the developer’s source code.

Statement-level metamutant As shown by Table A.1, our tool instruments statements by inserting an *if* statement around them. The *if* statement’s condition calls

PseudoSweep to determine whether to execute the contained statement. The expression includes the class name, element type and element number to enable PseudoSweep to identify it. Where moving a statement into an *if* block causes scoping issues and thus compilation concerns, the instrumentation uses scenario-specific tactics. For instance, variable declarations and return statements cannot change scope, so PseudoSweep uses default values so that the program still compiles.

Method-level metamutant We implement method-level instrumentation by inserting a conditional *if* statement around the method logic as demonstrated in Table A.1. We also addressed scoping issues at the method level by adding default values to relevant places to ensure the code could still compile.

The details of the mutants are recorded in class information files, enabling developers to identify where PseudoSweep placed the mutants in the original source code without requiring them to inspect the metamutant. The "Analyze" command also uses these files to identify the pseudo-tested code.

Methods such as simple getter/setter methods may be unimportant to test, therefore PseudoSweep is configurable to skip their instrumentation, with a view to extend this to other less-relevant statements, such as logging calls, in future work.

Due to the use of JavaParser [178], the projects that PseudoSweep can evaluate are limited to Java versions 8 to 12. The tool cannot evaluate code containing *var* type definitions, as type information is required to set default values. Currently, the tool supports JUnit 4 and 5 test suites, with the scope of adding compatibility with other frameworks. Deng et al.'s definitions of SDL [49] and the terms used in JavaParser guide PseudoSweep's definitions of a *statement*; therefore, our tool's use of this word may differ from other tools'.

After instrumenting the source code, the user must compile it with their configured build tool, including all their dependencies and test classes, before proceeding to the next phase.

A.2.2 Sweep

The main algorithm for evaluating the instrumented class files is similar to regular mutation analysis. Given a test method, PseudoSweep executes it three times, without any active mutations, to record the test results, times taken (to calculate timeout values) and the methods and statements executed. The tool executes the tests three times because a flaky test result could cause pseudo-tested elements to be missed if a test fails for reasons unrelated to the deleted element [154]. If a test fails at this stage, the tool discards it as it cannot be relied upon for testing the code. Secondly, PseudoSweep iterates through the list of executed elements, activating the relevant mutations individually and checking

for behaviour that differs from the recorded outcomes, specifically failing, throwing an exception or timing out. The test runs three times, attempting to induce one of these outcomes. If one of these occurs, PseudoSweep stops and moves on to the next element. It then records the test outcomes in the result files for analysis purposes. Due to internal thread tracking, projects containing threaded code may cause errors and unreliable results.

A.2.3 Analyze

After evaluating all the mutants, PseudoSweep uses the Analyze functionality to identify the not-covered, covered and pseudo-tested elements. The tool presents the lists of elements and relevant statistics (e.g., % pseudo-tested) in a *.json* file per Java class, as well as an overall project summary file. At this stage, the developer can look directly at the metamutant to identify the pseudo-tested elements, or invoke the Restore functionality to return the code to its original state and use the provided location information to begin addressing the pseudo-tested code.

A.2.4 Restore

The Restore functionality deletes the instrumented file and renames the original file back to *filename.java*. Although this is a manual step in the current prototype, future versions of the tool will internalise code restoration and provide visual coverage reports that display pseudo-tested elements.

A.3 Applying the PseudoSweep tool

We summarise how PseudoSweep has been evaluated in Chapter 3 and how it can be used with a case study and a demonstration. Full instructions on setting up the tool are available on GitHub [3].

A.3.1 Evaluation

This section summarises the results of the empirical evaluation in Chapter 3 that applied PseudoSweep to 27 real-world projects, including Apache projects *commons-codec* and *commons-cli* [129]. The study explored the frequency, weaknesses and causes of pseudo-tested statements in these Java projects. The study found that identifying only pseudo-tested methods would miss half of the pseudo-tested statements. PseudoSweep identified 350 pseudo-tested statements within methods required for the test suite to pass. The study also revealed that these statements obtained lower mutation scores, suggesting they

are less well-tested, with the leading causes of pseudo-tested statements being missing/-partial tests and assertions, which accounted for 86 of the 119 pseudo-tested statements sampled.

A.3.2 Case Study

Our study of the causes of pseudo-tested statements in 27 real-world Java projects identified many examples, including the *commons-codec.language.Caverphone2.encode* method in the *commons-codec* project that this section uses as a case study. The *encode* method encodes a source string into a CaverPhone 2.0 value as shown in Figure A.2. The method uses a sequence of *java.lang.String.replace(t, r)* and *java.lang.String.replaceAll(r,r)* method calls to substitute characters and regular expression patterns within the source string to produce CaverPhone 2.0 value. Twelve tests cover the method, achieving 100% statement coverage. This method is also "required" for the test suite to pass (i.e., it cannot be removed without causing tests to fail). However, PseudoSweep identified that of the 63 statements in the method, 23 were pseudo-tested. The first author manually confirmed this by deleting each statement and running the test suite. We provide full replication instructions in the replication GitHub Repository [1], including a link to the video presentation of the tool evaluating this Java class in the README. We will now look at how the tool identifies these statements.

Generating mutants

Given the source code of *commons-codec.language.Caverphone2.encode*, as shown in Listing A.2, PseudoSweep applies the statement operators described in Table A.1 to each statement. The statement-level operators also use *if* statements to execute the statement logic conditionally. For example, given Line 14 in Listing A.2, PseudoSweep applies the **SDSS** operator to produce the following instrumentation:

```
if (org.pseudosweep.I.exec("EXPRESSION", 9, "org.apache.commons.
    codec.language.Caverphone2", "sdl")) {
    txt = txt.replaceAll("^trough", "trou2f");
}
```

Executing mutants

During the Sweep command, PseudoSweep executes tests against mutants by recording the elements covered by each test, then running each test against each mutated version of the elements it covers.

Identifying pseudo-tested code

The Analyze command enables the user to further analyse sweep test results to reveal the pseudo-tested components. The developer has two choices at this stage: to Restore

```
1   @Override
2   public String encode(final String source) {
3       String txt = source;
4       if (SoundexUtils.isEmpty(txt)) {
5           return TEN_1;
6       }
7       txt = txt.toLowerCase(java.util.Locale.ENGLISH);
8       txt = txt.replaceAll("[^a-z]", "");
9       txt = txt.replaceAll("e$", "");
10      txt = txt.replaceAll("^cough", "cou2f");
11      txt = txt.replaceAll("^rough", "rou2f");
12      txt = txt.replaceAll("^tough", "tou2f");
13      txt = txt.replaceAll("^enough", "enou2f");
14      txt = txt.replaceAll("^trough", "trou2f");
15      txt = txt.replaceAll("^gn", "2n");
16      txt = txt.replaceAll("mb$", "m2");
17      txt = txt.replace("cq", "2q");
18      txt = txt.replace("ci", "si");
19      txt = txt.replace("ce", "se");
20      txt = txt.replace("cy", "sy");
21      txt = txt.replace("tch", "2ch");
22      txt = txt.replace("c", "k");
23      txt = txt.replace("q", "k");
24      txt = txt.replace("x", "k");
25      txt = txt.replace("v", "f");
26      txt = txt.replace("dg", "2g");
27      txt = txt.replace("tio", "sio");
28      txt = txt.replace("tia", "sia");
29      txt = txt.replace("d", "t");
30      txt = txt.replace("ph", "fh");
31      txt = txt.replace("b", "p");
32      txt = txt.replace("sh", "s2");
33      txt = txt.replace("z", "s");
34      txt = txt.replaceAll("^[aeiou]", "A");
35      txt = txt.replaceAll("[aeiou]", "3");
36      txt = txt.replace("j", "y");
37      txt = txt.replaceAll("^y3", "Y3");
38      txt = txt.replaceAll("^y", "A");
39      txt = txt.replace("y", "3");
40      txt = txt.replace("3gh3", "3kh3");
41      txt = txt.replace("gh", "22");
42      txt = txt.replace("g", "k");
43      txt = txt.replaceAll("s+", "S");
44      txt = txt.replaceAll("t+", "T");
45      txt = txt.replaceAll("p+", "P");
46      txt = txt.replaceAll("k+", "K");
47      txt = txt.replaceAll("f+", "F");
48      txt = txt.replaceAll("m+", "M");
49      txt = txt.replaceAll("n+", "N");
50      txt = txt.replace("w3", "W3");
51      txt = txt.replace("wh3", "Wh3");
52      txt = txt.replaceAll("w$", "3");
53      txt = txt.replace("w", "2");
54      txt = txt.replaceAll("^h", "A");
55      txt = txt.replace("h", "2");
56      txt = txt.replace("r3", "R3");
57      txt = txt.replaceAll("r$", "3");
58      txt = txt.replace("r", "2");
59      txt = txt.replace("l3", "L3");
60      txt = txt.replaceAll("l$", "3");
61      txt = txt.replace("l", "2");
62      txt = txt.replace("2", "");
63      txt = txt.replaceAll("3$", "A");
64      txt = txt.replace("3", "");
65      txt = txt + TEN_1;
66     return txt.substring(0, TEN_1.length());
67  }
```

Listing A.2: The *commons-codec.language.Caverphone2.encode* method (with blank space and comments removed for presentation purposes). Pseudo-tested statements identified by PseudoSweep are highlighted with grey boxes (© 2024 IEEE).

the code to the source code or to analyse the metamutant directly. The metamutant contains the element information displayed in the analysis file, enabling the user to use file searches to quickly locate the pseudo-tested code. Otherwise, they can restore the system to its source code and use the class information files to identify the locations of

pseudo-tested code elements. Once the user has found the pseudo-tested element in the source code, they can begin to analyse it. Using file search, the user can discover which test cases covered the element and manually analyse the deletion to decide their action plan for addressing the issue. According to the code comments, the statement on Line 14 in Listing A.2 is exclusive to the CaverPhone 2.0 specification. However, PseudoSweep can remove this statement without impacting test outcomes. Adding a test assertion such as:

```
assertEquals(encode("trough"), "TRF1111111")
```

to check the encoding of a string beginning with "trough" means this statement is required for the test suite to pass.

A.4 Conclusions

This chapter introduced PseudoSweep, a tool that detects pseudo-tested statements so that developers can identify areas of insufficient testing before committing resources to the costly process of mutation testing (**D1**). This chapter explained PseudoSweep's design and its approach to identifying pseudo-tested methods and statements in Java programs (**R1**). Finally, we demonstrated how a developer can use the output from our tool in their workflow, using an example from *commons-codec*.