How Effective are Coverage- and Diversity-Based Test Selection at Killing Stubborn Mutants?

Islam T. Elgendy
University of Sheffield

Robert M. Hierons
University of Sheffield

Phil McMinn
University of Sheffield

***Abstract*—Coverage- and diversity-based test selection are two prominent strategies for accelerating regression testing by deriving a smaller set of tests from an original test suite. While coverage-based approaches maximise fault detection, they require up-front code instrumentation and execution. Diversity-based approaches, on the other hand, select tests based on dissimilarity, but it has been found that they discover fewer faults. However, the specific characteristics of the faults detected by these two strategies remain unexplored. This raises critical questions: Does the superior raw performance of coverage-based approaches stem from merely finding "trivial" faults, and how do both strategies fare against subtle, hard-to-detect bugs? This is vital because optimising for raw fault counts alone may create a false sense of security, masking an inability to detect deeper faults that could present unique challenges to system reliability. Mutation analysis offers a means of studying this behaviour, simulating hard-to-find bugs by generating "stubborn" mutants — mutants that can only be detected by a small number of tests. We empirically investigated how coverage- and diversity-based approaches perform with mutants and real faults of varying stubbornness across seven Java projects. Results show that diversity-based selection often rivals or surpasses coverage for the most difficult mutants, requiring up to 46% fewer tests. We further show that tests killing only stubborn mutants constitute just 2–25% of a suite yet achieve full mutation adequacy and detect up to 88% of real faults, and that such mutants are disproportionately found in private, void, and deeply nested methods. These findings highlight stubborn mutants as efficient test targets and lightweight diversity-based selection as a scalable alternative to coverage, avoiding pre-execution and instrumentation costs.**

***Index Terms*—mutation testing, stubborn mutant, diversity-based testing, test case selection**

I. Introduction

Regression testing ensures that software modifications do not introduce new faults. As projects evolve, test suites grow in size and execution cost, often making it impractical to run every test after every change [32]. To address this, researchers have developed test selection strategies to execute only the tests deemed most valuable — for example, by finding the most faults, or faults that other tests are unlikely to detect. *Coverage-based* techniques address this problem by selecting tests that maximise coverage. While empirical studies show that they consistently find the most faults [29], [32], they require instrumentation of project code and pre-execution of the entire test suite to collect coverage information, which can be costly for large systems. *Diversity-based* approaches, in contrast, select tests that are most dissimilar to others in the test suite, using a similarity metric [10]. Typically, diversity-based approaches detect fewer faults, but are more lightweight, since diversity can be measured statically using the source code [21], [8] or bytecode [9] of the tests.

Despite these trade-offs, the *nature* of detected faults remains unexplored. For instance, while coverage-based techniques yield higher fault counts, they may primarily uncover trivial defects. Favouring raw counts over tests capable of exposing deeper, subtle faults risks creating a false sense of security and compromising the reliability of future system releases.

This paper is the first to investigate this issue, applying mutation analysis [17] to generate artificial faults of varying "difficulty". Particularly hard-to-detect faults are referred to as *stubborn* mutants. Figure 1 shows an example from the `Fraction` class in the widely-used Apache Commons Math library for Java, and one of our subjects used in our experimental analysis later in the paper, with three mutants depicted. The first mutant (M_1) affects line 5 and is detected ("killed") by almost all tests, because it reverses a rare overflow condition that throws an exception. The second (M_2) affects line 8. It affects the calculation of a value used in the condition, and is consequently more "difficult" to detect and is killed by fewer tests. Finally, the third (M_3) alters line 38, and is only killed by a very small number of tests. This mutant introduces an "off-by-one" error, and is particularly hard to detect. To do so, requires a very specific input combination that makes two values equal to one another, thereby exposing the difference between the mutant and the original version of the code.

The relationship between *test selection* and *mutant difficulty* has not been systematically examined. Existing studies mainly report overall mutation scores or fault detection rates, implicitly treating all mutants as equally important [21], [24], [9]. Consequently, it remains unclear whether coverage-based techniques perform well because they detect *many* faults or because they detect the *most difficult* ones. Clarifying this distinction is essential for assessing the true value of different selection strategies, especially in cost-sensitive settings such as regression testing and continuous integration. Because stubborn mutants correlate with real fault difficulty [19], [27], they also offer a conceptual bridge to understanding "hard-to-find" real faults (i.e., faults that are killed by very few tests). In this work, we extend this connection by analysing not only stubborn mutants but also real faults that are difficult to detect.

To address this gap, we analyse how test selection strategies perform across mutants of varying stubbornness. We compare coverage-based and diversity-based approaches on seven real-world Java projects from Defects4J [18], examining: (1) their

```
1  private Fraction(double value, double epsilon,
       int maxDenominator, int maxIterations) {
2    long overflow = Integer.MAX_VALUE;
3-4  double r0 = value; long a0 = (long)floor(r0);
5  - if (a0 > overflow) {
   + if (a0 <= overflow) { // M1
6      throw new FractionException(value, a0, 1l);
7    }
8  - if (abs(a0 - value) < epsilon) {
   + if (abs(a0 + value) < epsilon) { // M2
9      this.numerator= (int) a0;
10-11  this.denominator = 1; return;
12   }
     ...
23-26  long p2=...;long q2=...;int n=...;
27   do {
       ...
37     double convergent = (double)p2/(double)q2;
38 -   if(n<maxIterations && abs(convergent-value)
           > epsilon && q2<maxDenominator) {
   +   if(n<maxIterations && abs(convergent-value)
           > epsilon && q2<=maxDenominator) { // M3
       // ...
47   } while (...);
     ...
   }
```

Figure 1: A constructor of the `Fraction` class from the Apache Commons Math library, demonstrating three mutants, M1–M3. Omitted code sections are replaced with "...".

effectiveness in killing stubborn mutants; (2) the structural and semantic characteristics of these mutants; and (3) the practical value of the tests that kill them. Results show that diversity-based using bytecode information (i.e., the compiled form of tests) performs comparably or better than coverage-based selection, requiring 46% fewer tests on median to kill the most stubborn mutants. Also, diversity-based required 11–40% fewer tests than coverage-based selection to detect the hardest faults. Stubborn mutants are disproportionately found in private and `void` methods, deeply nested code, and are frequently produced by *method-level* mutation operators (e.g., `VoidMethodCall`). These findings provide new insight into where and how stubborn mutants occur and can guide the design of more effective mutation operators targeting high-utility faults. Finally, we show that tests killing only stubborn mutants constitute just 2–25% of a suite, yet achieve full mutation adequacy and detect up to 88% of real faults.

These findings provide the first empirical evidence that diversity-based test selection can effectively detect hard-to-find faults, while also deepening our understanding of their properties and practical significance. We provide a replication package [11] containing all data, source code, and instructions for reproducing our experiments. This paper makes the following contributions:

1) The first empirical study of how test selection strategies relate to fault difficulty, using stubborn mutants and hard-to-find real faults.
2) A comparison of coverage-based and diversity-based test selection on seven real-world projects, analysing effectiveness across mutant stubbornness levels.
3) An empirical characterisation of stubborn mutants' code locations, mutation operator types, and their practical relevance for mutation testing.

II. Background

A. Mutation Analysis

Mutation analysis is a fault-based technique that evaluates test suite effectiveness by introducing small syntactic changes, called *mutants*, into a program [17], simulating real developer faults. It relies on two hypotheses: the Competent Programmer Hypothesis (CPH) [6], which assumes programmers write nearly correct code with small deviations, and the Coupling Effect [25], which suggests that tests detecting simple errors also detect more complex ones.

A mutant is *killed* if at least one test fails on the mutated code, and *live* if all tests pass, indicating potential weaknesses in the developer test suite. Some mutants can never be killed; these are *equivalent mutants*, which are syntactically different but semantically identical to the original program.

B. Stubborn Mutants

In principle, the idea of a stubborn mutant is the same across different works, but the definition of these stubborn or hard-to-kill mutants varies. A stubborn mutant is one that is very hard to kill, while an equivalent mutant is impossible to kill. All mutants lie on a spectrum of "killability" from easy to kill to impossible to kill [31]. Stubborn mutants occupy the space immediately adjacent to equivalent mutants on this spectrum. They subsume easier-to-kill mutants [20], [23], making them a stronger indicator of test suite effectiveness, and their difficulty correlates more strongly with real fault detection [19], [27].

Papadakis et al. [27], Hernandez et al. [15], and Du et al. [7] consider a mutant stubborn if very few tests kill it. Hernandez et al. [15] introduced the rank-stubbornness model (RSM), which assigns each mutant a rank based on the number of killing test cases. We refer to RSM in Section III-A. Also, Lindström et al. [23] reported that a stubborn mutant is killed by only one or very few tests. We use a similar definition.

C. Test Selection Techniques

Test selection techniques aim to identify a subset of test cases from a larger suite that maximises fault detection while reducing execution cost [32]. These techniques are widely used in regression testing and continuous integration, where executing the full suite can be expensive.

Coverage-based techniques select tests based on the program elements (e.g., statements, branches, or methods) they exercise [29]. These approaches are *dynamic*, requiring pre-execution of the test suite to collect coverage information, which can be costly for large systems. Despite this, coverage-based selection has consistently been shown to achieve strong fault detection performance.

Diversity-based techniques, in contrast, select tests based on *dissimilarity* between test cases [10]. Dissimilarity is measured using a similarity metric (e.g., Euclidean distance, or Jaccard similarity) over program artefacts, such as test inputs or outputs. These methods are *static* and *lightweight*, as they do not

Table I: Example showing mutant stubbornness and kill scores

Tests	m_1	m_2	m_3	m_4	m_5	m_6	m_7
T_1	✓	✓	✗	✓	✗	✗	✓
T_2	✓	✗	✗	✓	✗	✗	✗
T_3	✓	✗	✗	✓	✓	✗	✓
T_4	✓	✓	✓	✗	✗	✓	✗
T_5	✓	✓	✗	✗	✗	✗	✗
T_6	✓	✗	✗	✓	✓	✓	✓
T_7	✗	✓	✗	✓	✗	✗	✓
Kill score	6	4	1	5	2	2	4

require test execution, making them substantially cheaper than coverage-based strategies [9].

III. Approach

This section outlines our approach to identifying and evaluating stubborn mutants and analyses how different test selection strategies detect them early. We present our model for identifying stubborn mutants and describe the test selection strategies employed in our study. Then, we assess the significance of the tests killing only stubborn mutants.

A. Stubborn Mutant Identification

Table I shows an example of seven tests and seven mutants. Each cell entry indicates whether a test T_i kills a mutant m_j (‘✓‘) or not (‘✗‘). As discussed in Section II-B, the Rank-Stubbornness Model (RSM) assigns a rank to each mutant based on how many tests kill it. However, the term rank may be misleading, as it can suggest that each mutant is assigned a contiguous ordinal position. Instead, we use the term **Kill Score**, representing the number of distinct test cases that kill each mutant, allowing multiple mutants to share the same score. The last row of Table I (*Kill Score*) shows these values. For instance, mutant m_1 is killed by six tests, indicating it is easy to detect, while m_3 is killed by only one test and m_5 and m_6 by two, making them harder to kill.

The Rank-Stubbornness Model (RSM) ranks mutants by their resistance to being killed but lacks a principled threshold for deciding which mutants are truly stubborn, relying instead on arbitrary cutoffs. Papadakis et al. [27] introduced fixed thresholds (e.g., mutants killed by no more than 5% or 2.5% of tests are considered stubborn). However, such rigid cutoffs can misclassify mutants, particularly in projects where even easy mutants are killed by a few tests. In such cases, many mutants may be incorrectly labelled as stubborn.

Stubbornness is inherently project-dependent: a score that denotes high resistance in one system may represent average difficulty in another. Thus, absolute thresholds yield inconsistent or misleading classifications across projects.

To address these issues, we propose the **Relative Stubbornness Thresholding Model (RSTM)**. RSTM introduces a user-defined relative threshold that identifies stubborn mutants based on their difficulty *relative to the easiest mutant* within the same project. This approach enables consistent, flexible, and project-aware identification of stubborn mutants. By adjusting the threshold, testers can control the sensitivity of stubbornness detection to suit their analysis needs.

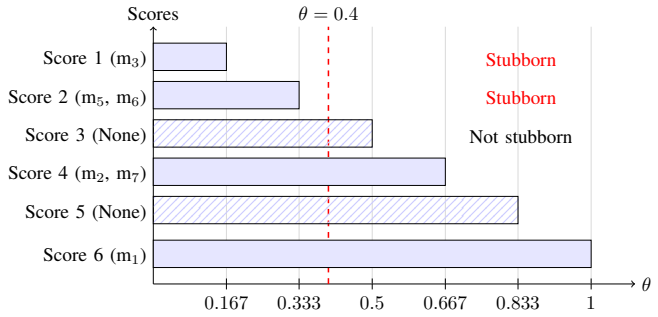


Figure 2: Illustration of RSTM: grouping mutants by threshold levels. Filled bands represent mutants present in the example; hatched bands (Score 3 and 5) are added for illustration but contain no mutants in this scenario.

For a set of n mutants and a test suite T, RSTM assigns each mutant m_i a Kill score $s_i \in \mathbb{N}$, which is the number of test cases in T that kill m_i. Let:

- $S = \{s_1, s_2, \ldots, s_n \mid 1 \leq s_i \leq |T|\}$ be the multiset of Kill scores assigned to each of the n killable mutants.
- $s_{\max}$ be the highest score value, corresponding to the easiest-to-kill mutant (i.e., killed by the most tests).
- $\theta \in (0, 1]$ be a user-defined threshold that determines the level of relative stubbornness.

A mutant m_i is classified as *stubborn* under the RSTM if:

$$s_i \leq \theta \cdot s_{\max}$$

Using the example from Table I, $s_{\max} = 6$ and if $\theta = 0.4$, then the threshold is $6 \times 0.4 = 2.4$, and any mutant with $s_i \leq 2.4$ is considered stubborn. So, m_3, m_5, and m_6 are stubborn mutants. If the stubbornness threshold is 0.25, then only m_3 is stubborn. RSTM enables flexible identification of relatively stubborn (i.e., harder-to-kill) mutants.

Figure 2 visualises this example. The y-axis shows the mutant scores (number of killing tests), while the x-axis represents the corresponding threshold levels (θ). Mutants are grouped into horizontal bands by score, and the θ line defines which groups fall below the threshold. For instance, with $\theta = 0.4$, mutants in groups with $\theta \leq 0.4$ (e.g., m_3, m_5, m_6) are classified as stubborn, while those in higher bands are not.

To extend this analysis from artificial faults to real faults, we applied the same RSTM logic to each real fault in our dataset. Specifically, we treated each real fault analogously to a mutant by considering the number of test cases that detect it. A real fault is classified as *hard-to-find* if the number of tests detecting it (s_i) falls below the relative stubbornness threshold (i.e., $s_i \leq \theta \cdot s_{\max}$), where $s_{\max}$ corresponds to the easiest-to-kill mutant within the same project. This approach enables a unified and project-aware characterisation of fault difficulty for both mutants and real faults, allowing us to directly compare how test selection strategies perform on hard-to-find faults in both categories.

B. Test Case Selection Approaches

We implemented six test case selection approaches, two of which are coverage-based and four are diversity-based.

The first coverage-based approach prioritises maximum statement coverage [29], hypothesising that stubborn mutants are killed by broadly-scoped tests. We used a greedy algorithm that iteratively selects the test case with the highest additional coverage until the mutant is killed. Since all mutants in our study are killable as identified by RSTM, this process always terminates. The second approach follows the same logic but prioritises minimum coverage. This allows us to investigate whether narrow, focused tests are more effective at killing stubborn mutants than those with a wider focus. By implementing these two opposing strategies, we can analyse which coverage-based property (breadth or focus) is more strongly associated with the ability to reveal stubborn mutants.

We implemented two state-of-the-art diversity-based selection approaches, Ledru [21] and FAST [24], motivated by the idea that stubborn mutants are hard to kill and may require diverse test inputs or behaviours to be triggered. Ledru is a pairwise similarity approach that selects tests incrementally, adding at each step the test that maximises its dissimilarity to the already selected set, while FAST approximates this process using some big data techniques, such as Shingling, Minhashing, and Locality-Sensitive Hashing, to reduce computational cost. For each approach, we used two types of diversity information. Textual diversity (Ledru-Text [21] and FAST-Text [24]) measures differences in the text of test cases, capturing variations in inputs and method sequences. Bytecode diversity [9] (Ledru-Bytecode and FAST-Bytecode) measures differences in compiled test instructions, capturing variations in actual program behaviours while being more compact and efficient. Comparing these four variants enables us to assess how both the algorithm and the choice of diversity representation impact the ability to kill stubborn mutants. Due to space limitations, more details on the diversity-based algorithms are provided by Elgendy et al. [9].

C. Effectiveness of Stubborn-Mutant-Killing Tests

We want to identify the impact of stubborn mutants on test suite effectiveness and real fault detection. An interesting observation from Table I, is that some mutants *subsume* others. Just et al. [19] found that subsuming mutants have higher utility, as they are less likely to be equivalent or trivial. Other works found that stubborn mutants are more likely to subsume other mutants [23]. Killing m_5 (e.g., by T_3 or T_6) also results in the killing of m_1, m_4, and m_7. Similarly, killing m_3 (via T_4) also kills m_1, m_2, and m_6. This subsumption effect suggests that some stubborn mutants may be more valuable than their apparent rarity implies, because they represent faults whose detection indirectly kills multiple other mutants.

Returning to the example, selecting T_4 (which kills m_3) and either T_3 or T_6 (which kill m_5) is sufficient to kill all seven mutants. This demonstrates the potential utility of stubborn mutants in test suite optimisation. Focusing on tests that can kill the stubborn mutants could not only reduce the test suite size but also improve overall test effectiveness by collapsing redundant mutant coverage.

Table II: The test subjects used in our work

ID	#Versions	#Mutants	# Test Cases			
			Randoop	EvoSuite	Developer	Total
Cli	5	786	7425	279	242	7946
Compress	7	1051	10769	133	38	10940
Csv	10	767	15050	260	550	15860
Jsoup	16	767	20575	278	632	21485
Lang	15	4136	25893	1103	688	27684
Math	42	10986	27235	962	1612	29809
Time	2	268	499	172	69	740
Total	97	18826	107446	3187	3831	114464

In our approach, we identify the set of tests that kill the set of stubborn mutants, denoted as (T_{Stb}). We analyse the impact of killing the stubborn mutants by analysing the mutation score achieved by T_{Stb}. We can identify whether these tests can reveal real faults in the program by calculating the *Fault Reveal Rate* (FRR), which is the proportion of error-revealing tests that appear in T_{Stb}. The set of error-revealing tests is denoted as (T_E), which we identify using the Defects4J benchmark [18]. The FRR value is in the range $[0, 1]$ and is calculated as $FRR = |T_{Stb} \cap T_E|/|T_E|$, where $FRR = 1$ indicates that all error-revealing tests are included, and $FRR = 0$ that none are.

IV. Evaluation

To validate the potential of diversity-based test selection as a lightweight means of killing stubborn mutants and to deepen our understanding of their properties, this study is guided by the following research questions:

- **RQ1:** How do different test selection strategies (coverage-based and textual/bytecode diversity-based) compare in their ability to kill stubborn mutants early under different relative stubbornness thresholds?
- **RQ2:** How effectively do the selection strategies detect hard-to-find real faults, and do their behaviours mirror those observed for stubborn mutants?
- **RQ3:** What mutation adequacy do tests that kill stubborn mutants achieve, and how does this relate to real fault detection?
- **RQ4:** Where do stubborn mutants usually occur, and what are their characteristics?

A. Subjects

Our experiments utilised Java projects from Defects4J [18]. From the benchmark, we selected seven projects where both JaCoCo [16] and PIT [3] executed correctly. Projects with persistent execution failures were excluded. The subjects are listed in Table II, totalling 97 versions. We used PIT [3], a common tool in academia and industry [33], to generate mutants. For each version, we target the specific classes identified by Defects4J as containing real faults.

To ensure our study included a sufficient number of stubborn mutants, we required large and diverse test suites capable of revealing subtle or subsuming behaviours. For this reason, we augmented developer-written tests with automatically generated tests from Randoop [26] and EvoSuite [13]. Larger and more comprehensive test suites increase the likelihood of killing

a broad range of mutants, including those that are harder to detect. As noted by Just et al. [18], more complete test suites provide a better approximation when identifying dominator sets (mutants that subsume others), a concept closely related to stubborn mutants.

B. Methodology

We developed a tool to automate stubborn mutant identification and test selection strategies. We explain data collection, test selection approaches, and stubborn mutant analysis.

1) Data Collection: We used publicly available data [12] containing error-revealing tests, statement coverage per test case, similarity, and mutation information for the seven projects reported in Table II. Full details on the data structure and format are provided in the replication package [11].

2) Test Case Selection Approaches: As we explained in Section III-A, we used the RSTM model to identify the stubborn mutants. In our experiments, we investigated different thresholds, which are 0.75, 0.5, 0.25, 0.1, 0.05, 0.01, 0.005, 0.003, 0.002, and 0.001. Our tool implemented the six coverage and diversity-based test selection approaches explained in Section III-B. For each stubborn mutant, the tool records the number of the first test case to kill that mutant.

3) Effectiveness of Stubborn-Mutant-Killing Tests: For all stubbornness thresholds, the tool determined T_{Stb} (i.e., the set of tests that kill the set of stubborn mutants). Then, it recorded the total mutation score, the mutation score achieved only by T_{Stb}, the FRR values, the size of the full test suite, and the size of T_{Stb}. To establish a baseline, we also generated random test subsets ($\mathcal{R}$) of the same size as T_{Stb}. Each random selection was repeated 31 times, and the results were averaged to reduce variance. This comparison with $\mathcal{R}$ allows us to distinguish the specific effectiveness of stubborn-mutant-killing tests from effects that could arise merely due to test suite size.

Our tool averaged values across each project's versions and counted the number of times a *perfect FRR* (i.e., a value of 1 indicating all error-revealing tests are selected) is achieved.

4) Stubborn Mutant Characteristics: We analysed some properties of the most stubborn mutants in each project. The most stubborn mutants have scores ranging from 1 (i.e., killed by only one test) to 5 (i.e., killed by five tests).

For each stubborn mutant, the tool identified the source file and line number where the mutant occurred. The tool parsed the abstract syntax tree to identify the type of the line (e.g., an `if` statement, a `return` statement, a method invocation, etc). The tool also recorded the nesting level of that line. In addition, the tool identified the method containing the mutated line and recorded the access modifier (i.e., `private`, `protected`, `public`), whether it is a `void` or non-`void` method, and whether it is a `static` or instance method. Finally, the tool recorded the mutation operator that generated the mutant. All this information was used to give us an idea of what might cause a mutant to be stubborn.

C. Threats to Validity

Construct validity: Fault difficulty is defined in our study as the number of test cases required to kill a mutant. We acknowledge that a mutant may appear difficult, not because it is inherently subtle, but because the affected code region has received less testing attention. To reduce this risk, we augmented developer-written tests with automatically generated tests that systematically explore behaviour and are not influenced by developers' decisions, lowering the likelihood that stubborn mutants are artefacts of limited manual testing. A related concern is whether auto-generated tests, which often share repetitive naming and structures, might artificially bias diversity-based metrics. However, our empirical data shows that diversity-based algorithms selected a balanced mixture of both developer and auto-generated tests. This suggests that the textual differences between test sources did not disproportionately influence the metrics or skew the selection results.

Internal validity: Results could be influenced by how stubborn mutants are identified via RSTM. We also applied the fixed-threshold method [27], observing negligible differences, indicating our findings are robust. Detailed comparisons are available in our replication package [11].

External validity: Findings may not generalise beyond our subjects. We mitigated this by evaluating seven diverse Java projects from Defects4J, covering 97 versions and 67 classes, reporting both median results and variability (e.g., standard deviation) with statistical significance (p-values, effect sizes). Project-level results are included in the replication package.

Reliability: Reproducibility is critical. We provide a comprehensive replication package [11] with source code, scripts, datasets, and detailed instructions. Methodology and evaluation metrics are fully documented in this paper.

V. Results

A. RQ1: Test case selection strategies to kill stubborn mutants

Table III shows our analysis of the test selection strategies in killing the stubborn mutants for all projects in our study. The first column shows the stubbornness threshold levels, and the second column shows the number of stubborn mutants that fall in that threshold. The rest of the table shows the median (Med) and standard deviation (SD) of the first test to kill stubborn mutants for every test selection strategy used in our study. The table presents how quickly different test selection strategies are able to kill stubborn mutants, measured by the median number of selected tests needed. Lower medians are better, indicating faster detection. The bolded values highlight the strategy with the lowest median at each threshold level, i.e., the most efficient in killing stubborn mutants early. The relatively large standard deviation values observed across thresholds indicate substantial variability in first-kill positions across subjects and mutants. This suggests that while median trends are informative, performance can vary considerably depending on project characteristics and mutant distribution.

Maximum coverage (Max-Cov) is the best-performing strategy at low stubbornness levels, and from threshold 0.005, it falls behind FAST-Bytecode. Minimum coverage (Min-Cov) is consistently the worst strategy, indicating that narrowly focused tests have no utility in killing stubborn mutants. FAST-Text

Table III: Median and standard deviation of the first test to kill stubborn mutants per selection strategy for all projects

Threshold Level	#Stb Mutants	Ledru-Text		Ledru-Bytecode		FAST-Text		FAST-Bytecode		Max-Cov		Min-Cov		Max-Cov/FAST-Byte	
		Med	SD	Med	SD	Med	SD	Med	SD	Med	SD	Med	SD	p-value	$\mathbf{A_{12}}$
0.75	13196	16.0	332.3	14.0	253.1	50.0	599.8	20.0	330.5	**5.0**	308.0	444.0	686.7	<**0.0001**	0.64 (Medium)
0.50	12690	17.0	337.5	16.0	257.2	55.0	605.7	21.0	336.3	**6.0**	313.2	446.0	691.5	<**0.0001**	0.64 (Medium)
0.25	11111	25.0	355.8	21.0	271.9	67.0	629.7	24.0	354.6	**9.0**	331.5	511.0	706.0	<**0.0001**	0.63 (Small)
0.10	8633	38.0	393.3	30.0	300.9	111.0	676.6	33.0	391.7	**13.0**	368.5	626.0	729.4	<**0.0001**	0.61 (Small)
0.05	6772	55.0	427.2	40.0	328.8	167.0	714.0	39.0	423.9	**17.5**	395.4	716.0	737.4	<**0.0001**	0.60 (Small)
0.01	2649	175.0	539.7	105.0	446.2	539.0	829.2	65.0	558.8	**57.0**	497.7	857.0	704.2	<**0.0001**	0.55 (Small)
0.005	1564	244.0	621.8	157.0	482.2	714.0	836.1	**44.0**	661.0	63.5	560.7	934.5	717.0	**0.0004**	0.54 (Small)
0.003	1079	256.0	640.0	179.0	499.9	730.0	852.2	**42.0**	720.9	87.0	615.1	905.0	695.6	0.6695	0.51 (None)
0.002	739	361.0	695.7	235.0	549.8	758.0	914.3	**61.5**	828.1	128.0	688.7	1006.0	741.7	0.5282	0.51 (None)
0.001	236	408.0	853.5	288.5	724.9	1664.0	1107.1	**217.5**	1095.8	404.0	862.6	1601.0	814.4	0.7467	0.49 (None)

performed the worst out of all diversity-based strategies. Ledru-Text and Ledru-Bytecode are very close to each other, and their performance is relatively close to that of maximum coverage at broad stubbornness thresholds, and slightly better than FAST-Bytecode. However, after threshold 0.05, they fall behind FAST-Bytecode, and the gap gets bigger with higher threshold levels. FAST-Bytecode was reported by far the best performing in terms of runtime [9]. That makes FAST-Bytecode the best candidate of the diversity-based strategies overall.

To assess statistical significance, we conducted Mann-Whitney U tests and calculated Vargha-Delaney effect sizes. With 15 pairwise comparisons across six strategies, we focus our discussion on the two best-performing approaches, FAST-Bytecode and Max-Cov, which also represent the two families of approaches considered in this study: static lightweight diversity-based and dynamic coverage-based.

Comparison between FAST-Bytecode and Max-Cov: Although Max-Cov achieves the lowest median number of tests needed to kill stubborn mutants at higher threshold levels (e.g., θ = 0.75, median = 5), the median values achieved by FAST-Bytecode are close in magnitude. For example, at threshold levels of 0.75, 0.50, and 0.25, FAST-Bytecode yields medians of 20, 21, and 24, respectively. This is significant because FAST-Bytecode is a lightweight strategy, requiring no prior execution of the test suite, whereas Max-Cov depends on dynamic coverage information. The closeness of their performance suggests that a lightweight diversity-based approach (FAST-Bytecode) can be competitive with heavier, execution-dependent approaches in the early stages of test selection.

The last two columns in Table III show the statistical comparison between FAST-Bytecode and Max-Cov using Mann-Whitney U test and the Vargha-Delaney A_{12} effect size. The results show that the performance of Max-Cov is statistically significantly better than FAST-Bytecode for thresholds 0.75 through 0.01, with p-values $\leq$ 0.001, and the effect size shows a medium to small effect. For stricter stubbornness (e.g., $\theta \leq$ 0.005), the statistical difference diminishes, with p-values increasing and A_{12} effect sizes nearing 0.5, implying no practical performance advantage of Max-Cov over FAST-Byte. In fact, from threshold 0.005–0.001, FAST-Bytecode outperforms Max-Cov in terms of median, though the difference is not statistically significant.

Conclusion for RQ1: Our findings indicate that while Max-Cov demonstrates strong performance at broader stubbornness thresholds by quickly killing mutants with fewer tests, its advantage decreases as the threshold becomes stricter. FAST-Bytecode is lightweight and does not require pre-execution of the test suite and consistently approaches Max-Cov in early-stage effectiveness and eventually surpasses it in the stricter thresholds (e.g., $\leq$ 0.005). FAST-Bytecode has up to 46% fewer tests required at the strictest threshold than Max-Cov.

B. RQ2: Test case selection strategies to detect hard real faults

Table IV shows our analysis of the test selection strategies in detecting the hard-to-find real faults for all projects. Across all thresholds, the behavioural trends closely mirror those observed for stubborn mutants. At broader thresholds (0.75–0.10), Max-Cov consistently identifies and kills faults earliest, with median first-kill positions between 14–21 tests. FAST-Bytecode remains competitive in this region and is consistently the second-best performer, often within 15–20 additional tests.

As the threshold becomes stricter, focusing on the hardest faults, the relative performance shifts. From $\theta \leq 0.01$, FAST-Bytecode surpasses Max-Cov, detecting the hardest real faults earlier (e.g., 58 vs. 96 tests at $\theta = 0.01$). This matches the pattern previously observed for stubborn mutants, where diversity-based selection becomes increasingly advantageous as fault difficulty increases.

However, unlike the stubborn-mutant results, these differences for real faults are not statistically significant across any threshold (all p-values ≥ 0.07), and the effect sizes remain small or negligible. Thus, while the direction of the trend is consistent, diversity-based selection gains strength as faults become harder to detect, the empirical support is weaker due to the limited number of real faults per threshold and the high standard deviation values.

Conclusion for RQ2: The effectiveness of selection strategies on hard-to-find real faults mirrors the patterns observed with stubborn mutants: Max-Cov performs best

Table IV: Median and standard deviation of the first test to kill hard-faults per selection strategy for all projects

Threshold Level	#Hard Faults	Ledru-Text		Ledru-Bytecode		FAST-Text		FAST-Bytecode		Max-Cov		Min-Cov		Max-Cov/FAST-Byte	
		Med	SD	Med	SD	Med	SD	Med	SD	Med	SD	Med	SD	**p-value**	$\mathbf{A}_{12}$
0.75	94	119.5	523.1	62.5	445.9	96.0	897.8	31.0	523.5	**14.5**	521.9	492.5	799.5	0.0781	0.57 (Small)
0.50	93	129.0	525.1	63.0	447.4	110.5	900.5	32.0	525.9	**14.0**	524.1	518.0	803.0	0.0718	0.58 (Small)
0.25	91	139.0	528.3	66.0	450.5	136.0	905.8	33.5	530.7	**15.0**	528.4	536.0	804.0	0.0795	0.58 (Small)
0.10	86	150.0	538.2	67.5	454.7	151.0	920.5	36.0	543.8	**16.5**	540.0	593.0	793.1	0.1159	0.57 (Small)
0.05	78	178.5	555.0	87.0	466.9	190.0	933.9	43.0	566.9	**21.0**	560.3	605.5	794.3	0.2220	0.56 (Small)
0.01	53	318.0	609.6	149.0	528.0	388.5	1028.9	**58.0**	664.3	96.0	639.8	807.0	814.5	0.9795	0.50 (None)
0.005	38	445.0	661.8	304.0	582.4	768.0	1087.2	**119.0**	742.1	171.5	709.2	884.0	832.2	0.7504	0.48 (None)
0.003	32	504.0	706.9	304.0	614.1	920.0	1114.3	**189.0**	788.7	302.5	743.1	947.5	865.4	0.7053	0.47 (None)
0.002	23	513.0	722.9	**272.0**	672.7	1003.0	1158.2	284.0	858.7	454.0	729.4	1011.0	890.0	0.6133	0.46 (Small)
0.001	8	859.0	905.6	1046.5	843.4	2508.5	1504.5	**804.5**	1112.8	908.0	849.8	1783.5	1048.8	0.8785	0.47 (None)

at broader thresholds, while FAST-Bytecode catches up and surpasses Max-Cov at stricter thresholds. However, due to a smaller sample and higher variability in detection cost, these differences are not statistically significant, and effect sizes are small to negligible.

C. RQ3: Effectiveness of Stubborn-Mutant-Killing Tests

Table V summarises the effectiveness of selecting tests that target only stubborn mutants (T_{Stb}) at different stubbornness thresholds across all projects. Although our setup considered thresholds from 0.75 down to 0.001, we report only results for thresholds $\geq$0.05. At lower thresholds (0.01–0.001), the number of stubborn mutants was too small to produce meaningful or generalisable results.Thus, for clarity and space, we focus on thresholds yielding consistent, non-trivial results across projects. Full results, including lower thresholds, are available in our replication package [11]. The first column lists the project name, average test suite size, and mutation score (MS) per version. The second column shows the threshold level (TL) used to identify stubborn mutants via RSTM. Columns three and four report the average MS achieved by T_{Stb} and $\mathcal{R}$, a random baseline of equal size. The next two columns show the proportion of versions in which T_{Stb} and $\mathcal{R}$ achieve the same MS as the full suite ("Perfect" MS). Subsequent columns present the average fault-revealing rate (FRR) and the proportion of versions with perfect FRR (i.e., FRR = 1.0). The final column lists the average size of T_{Stb} (equal to $\mathcal{R}$).

Across all projects, targeting only stubborn mutants (T_{Stb}) dramatically reduces test suite size. For example, average test counts drop from over 1600 in `Cli` and `Compress` to fewer than 150, and below 50 in `Csv` and `Jsoup`. At a threshold level of 0.75, T_{Stb} consistently achieves high mutation scores, typically matching the full suite with "Perfect MS" near 1.0 across most projects. In contrast, the random baseline $\mathcal{R}$ shows substantially lower MS and fault-revealing performance, with "Perfect MS" and "Perfect FRR" values remaining zero across all thresholds. This confirms that the strong performance of T_{Stb} arises from targeting stubborn mutants rather than from random test reduction.

In terms of real fault detection, FRR varies across projects. Four of the seven subjects (`Csv`, `Jsoup`, `Lang`, and `Math`) show moderate to high FRR at threshold 0.75, indicating that many real fault-revealing tests are captured within T_{Stb}. In contrast, `Cli` and `Compress` exhibit consistently low FRR, and the `Time` project highlights a limitation: none of its real fault-revealing tests was included in T_{Stb} across the two evaluated versions (FRR = 0). However, these three projects contain relatively few real faults (five, seven, and two for `Cli`, `Compress`, and `Time`, respectively), limiting the representativeness of their FRR outcomes.

A decline in FRR is therefore most evident in projects with sparse real fault data. Conversely, systems with larger fault sets exhibit consistently higher FRR, suggesting that T_{Stb} is more effective when sufficient real faults are available for evaluation.

Overall, the results show that stubborn mutants serve as valuable indicators of test effectiveness. Focusing on tests that detect them—particularly early in testing—can improve both adequacy and cost-efficiency. Empirically, a threshold of 0.75 offers a strong balance between FRR and suite size across all studied projects. We thus recommend 0.75 as a practical default, while allowing practitioners to adjust it according to project scale and testing goals.

Conclusion for RQ3: Stubborn mutants effectively guide targeted test selection. At moderate relative thresholds (e.g., 0.75), focusing only on stubborn mutants achieves high mutation adequacy and strong FRR while substantially reducing test suite size. Across our subjects, tests that killed only stubborn mutants comprised just 2–25% of the full suite, yet achieved perfect mutation scores in all but one project. In six of seven cases, these subsets reached average real fault FRR values between 63% and 88%, showing that many fault-revealing tests are included when focusing on stubborn mutants. By contrast, randomly selected subsets of equal size consistently underperformed—yielding lower mutation scores, lower FRR, and never achieving perfect MS or FRR.

D. RQ4: Characteristics of stubbornness

To examine whether certain code characteristics are associated with stubborn mutants, we measured the proportion of stubborn mutants with each characteristic compared to all mutants and performed Chi-Square tests of independence [28].

Table V: Effectiveness analysis of stubborn mutants for all projects

Project (Avg Size / Avg MS)	TL	Avg MS		Perfect MS		Avg FRR		Perfect FRR		Avg Size
		T_{Stb}	$\mathcal{R}$	T_{Stb}	$\mathcal{R}$	T_{Stb}	$\mathcal{R}$	T_{Stb}	$\mathcal{R}$	
Cli (1738 / 0.9)	0.75	0.9	0.85	1.0	0.0	0.68	0.40	0.5	0.0	141
	0.50	0.9	0.84	1.0	0.0	0.68	0.38	0.5	0.0	136
	0.25	0.9	0.82	1.0	0.0	0.64	0.37	0.3	0.0	125
	0.10	0.9	0.76	1.0	0.0	0.63	0.24	0.0	0.0	107
	0.05	0.9	0.73	1.0	0.0	0.5	0.16	0.0	0.0	95
Compress (1617 / 0.72)	0.75	0.72	0.54	1.0	0.0	0.63	0.33	0.2	0.0	112
	0.50	0.72	0.56	1.0	0.0	0.63	0.33	0.2	0.0	108
	0.25	0.70	0.46	0.6	0.0	0.63	0.24	0.2	0.0	104
	0.10	0.69	0.34	0.4	0.0	0.43	0.06	0.2	0.0	98
	0.05	0.69	0.31	0.4	0.0	0.43	0.03	0.2	0.0	97
Csv (1902 / 0.69)	0.75	0.69	0.64	0.8	0.0	0.73	0.38	0.6	0.0	32
	0.50	0.69	0.63	0.8	0.0	0.73	0.38	0.6	0.0	32
	0.25	0.69	0.62	0.6	0.0	0.53	0.27	0.4	0.0	27
	0.10	0.68	0.50	0.6	0.0	0.53	0.06	0.4	0.0	20
	0.05	0.63	0.44	0.4	0.0	0.53	0.02	0.4	0.0	19
Jsoup (1798 / 0.86)	0.75	0.83	0.67	0.9	0.1	0.82	0.31	0.8	0.0	51
	0.50	0.83	0.67	0.9	0.0	0.82	0.27	0.8	0.0	49
	0.25	0.83	0.56	0.9	0.0	0.72	0.15	0.7	0.0	47
	0.10	0.83	0.54	0.9	0.0	0.72	0.11	0.7	0.0	43
	0.05	0.79	0.49	0.8	0.0	0.6	0.06	0.5	0.0	40
Lang (1811 / 0.84)	0.75	0.84	0.77	1.00	0.27	0.88	0.46	0.82	0.0	268
	0.50	0.77	0.68	0.82	0.18	0.78	0.39	0.73	0.0	259
	0.25	0.76	0.67	0.73	0.18	0.72	0.37	0.64	0.0	246
	0.10	0.67	0.57	0.64	0.09	0.67	0.25	0.64	0.0	202
	0.05	0.58	0.50	0.45	0.00	0.58	0.16	0.55	0.0	155
Math (625 / 0.82)	0.75	0.82	0.73	0.94	0.10	0.84	0.43	0.74	0.0	157
	0.50	0.79	0.66	0.90	0.06	0.81	0.38	0.71	0.0	151
	0.25	0.78	0.64	0.84	0.06	0.77	0.32	0.68	0.0	125
	0.10	0.73	0.57	0.61	0.00	0.69	0.20	0.61	0.0	90
	0.05	0.64	0.44	0.35	0.00	0.54	0.11	0.48	0.0	69
Time (370 / 0.8)	0.75	0.80	0.72	1.0	0.0	0.0	0.0	0.0	0.0	102
	0.50	0.80	0.70	1.0	0.0	0.0	0.0	0.0	0.0	98
	0.25	0.80	0.65	1.0	0.0	0.0	0.0	0.0	0.0	83
	0.10	0.80	0.62	1.0	0.0	0.0	0.0	0.0	0.0	64
	0.05	0.78	0.50	0.5	0.0	0.0	0.0	0.0	0.0	34

Categories included access modifiers, method types, node types, mutation operators, and nesting levels, with count data indicating how many mutants belonged to each stubbornness score (Score1–Score5). Chi-Square tests assess whether a category is significantly associated with being a top-score mutant. We report both statistical significance (p-value) and effect size via Cramér's V [4] (0.1 = small, 0.3 = medium, 0.5 = large effect).

Since high proportions do not necessarily imply statistical significance, we also use standardised residuals (z-scores) to measure deviations from expected counts, which account for sample size. A higher z-score indicates a stronger over- or underrepresentation, accounting for sample size, with $|z| > 2$ considered significant.

Table VI shows proportions and statistical results for each characteristic, and Figure 3 visualises the distributions.

Prior work [33], [31] examined mutation score, code testability, and operator class correlations with stubborn mutants. The mutant operator class ABS is not included in PIT's default operators, and some PIT operators are method-level, beyond the five general classes. It would be interesting to explore links between code observability and PIT operators in future work.

Across all stubbornness scores, Chi-Square tests confirmed significant associations ($p < 0.001$, Table VI), though effect sizes were small (Cramér's V: 0.08–0.13), indicating subtle but meaningful trends.

1) Access Modifier: Stubborn mutants, particularly at higher scores, are more likely in *private* methods and less likely in *public* or *protected* ones (Figure 3a). Private methods show strong overrepresentation in top scores (e.g., Score1 $z = 12.0$), while protected methods are consistently underrepresented. The reduced visibility may limit observability, making mutants harder to detect, consistent with Zhu et al. [33]'s recommendation to refactor private methods to increase test effectiveness.

2) Method Type: Void methods are more prone to stubborn mutants across all scores, particularly Score1 ($z = 10.6$), while *non-void* methods are underrepresented, likely because return values aid assertion-based testing. Constructors show moderate overrepresentation, but the trend is strongest for void methods (Figure 3a). This aligns with Zhu et al. [33]'s suggestion to refactor void methods to improve testability.

3) Mutation Operator: Some operators disproportionately produce stubborn mutants. *VoidMethodCallMutator*, *MathMutator*, and *EmptyObjectReturnValsMutator* are consistently overrepresented, suggesting subtle semantic changes that evade detection, while *NegateConditionalsMutator* and *NullReturnValsMutator* are underrepresented (Figure 3b). Although Yao et al. [31] highlighted LCR as prone to stubborn mutants, we found *BooleanTrueReturnValsMutator* and *BooleanFalseReturnValsMutator* (both LCR) near baseline ($z = -0.5$), aligning with expectations.

4) Nesting Level: Mutants in deeper code (levels 3–5) are more likely to be stubborn, peaking at level 3 for Score1 ($z = 5.8$), while flat code (level 0) is underrepresented, supporting the idea that structural complexity increases mutation resilience. Higher nesting levels beyond 3 are not overrepresented, likely due to small sample sizes (Figure 3c).

5) Node Type: Stubborn mutants cluster in specific constructs. *MethodInvocation* and *BinaryOperation* nodes are consistently overrepresented, especially at higher scores (e.g., *MethodInvocation* Score1: $z = 8.8$), whereas control-flow nodes (*IfStatement*, *ReturnStatement*, *ForStatement*) are underrepresented, suggesting mutations in branching logic are easier to detect (Figure 3d).

While our analysis shows strong associations between certain code characteristics and stubborn mutants, explaining why they occur preferentially in some constructs is beyond this study and left for future work.

Conclusion for RQ4: Our results demonstrate clear links between code characteristics and mutant stubbornness: **Access modifiers:** *Private* methods are more likely to contain stubborn mutants, with over-representation decreasing from 12 (Score1) to 3.3 (Score5), likely due to limited visibility reducing test observability. **Method return types:**

Table VI: Proportions and statistical analysis of each category across the top five scored subsets. Bold values indicate the highest proportions and standardised residuals (z-scores) within each category for the corresponding subset. p denotes the p-value, C is Cramér's V (effect size), z represents standardised residuals, and % indicates the proportion of the subset. An asterisk (*) denotes $p < 0.001$.

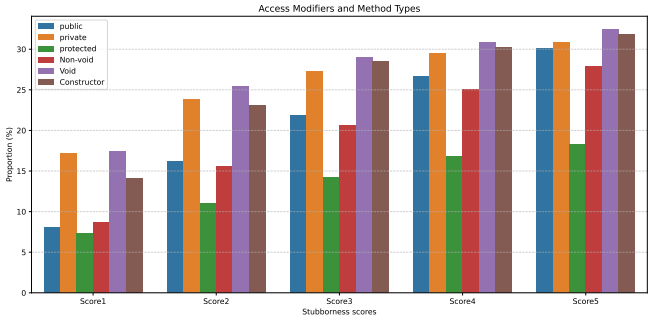
Category	Value	Score1				Score2				Score3				Score4				Score5			
		%	z	p	C	%	z	p	C	%	z	p	C	%	z	p	C	%	z	p	C
Access Modifier	public	8.1	-7.0			16.2	-3.6			21.9	-1.5			26.7	0.3			30.2	1.7		
	private	**17.2**	**12.0**	*	0.13	**23.8**	**8.5**	*	0.11	**27.2**	**5.8**	*	0.09	**29.5**	**3.6**	*	0.08	**30.8**	**2.1**	*	0.08
	protected	7.4	-4.0			11.0	-6.3			14.3	-6.7			16.9	-7.1			18.3	-7.6		
	package-private	11.0	0.1			19.4	0.7			24.6	0.8			28.5	0.8			30.6	0.6		
Method Type	Non-void	8.7	-6.3			15.6	-5.7			20.6	-4.4			25.1	-2.8			27.9	-2.1		
	Void	**17.4**	**10.6**	*	0.12	**25.4**	**9.3**	*	0.11	**29.0**	**6.9**	*	0.09	**30.8**	**4.5**	*	0.06	**32.4**	**3.3**	*	0.04
	Constructor	14.1	2.1			23.1	2.6			28.5	2.6			30.2	1.6			31.9	1.1		
Mutator	Negate-Conditionals	8.6	-4.5			14.6	-5.3			19.7	-4.2			24.3	-2.9			28.1	-1.1		
	Math	14.0	5.8			22.8	**6.6**			27.3	**5.7**			30.7	4.7			32.4	3.7		
	Conditionals-Boundary	7.8	-3.6			12.3	-5.3			14.6	-6.6			16.9	-7.3			18.4	-7.7		
	NullReturnVals	5.8	-4.8			12.6	-4.0			16.8	-3.9			19.1	-4.5			20.4	-5.0		
	VoidMethodCall	19.2	**7.4**			26.9	**6.1**			32.1	**5.7**			**36.0**	**5.4**			37.9	-3.1		
	Primitive-ReturnsCall	10.0	-0.6	*	0.12	16.3	-0.9	*	0.13	23.6	0.4	*	0.13	27.8	0.6	*	0.13	29.9	0.4	*	0.13
	EmptyObject-ReturnVals	8.3	-1.5			26.4	3.9			**33.2**	4.3			**36.5**	3.8			**41.5**	**4.5**		
	BooleanTrue-ReturnVals	9.9	-0.5			16.5	-0.7			22.4	-0.1			29.0	0.9			30.1	0.4		
	Increments	**21.3**	4.5			**27.4**	3.1			30.0	2.1			32.5	1.6			35.5	1.7		
	BooleanFalse-ReturnVals	9.5	-0.5			20.1	0.6			27.8	1.4			**36.7**	2.6			40.2	2.7		
	InvertNegs	18.0	2.4			21.3	0.9			27.9	1.2			33.6	1.5			33.6	0.9		
Nesting Level	0	8.3	-6.2			13.3	-8.8			17.6	-8.6			21.0	-8.4			23.6	-7.9		
	1	10.8	0.1			20.1	3.0			25.8	3.8			29.6	3.7			31.9	3.2		
	2	14.1	4.0			24.3	**5.9**			29.0	**5.2**			32.4	**4.6**			35.2	**4.6**		
	3	17.6	**5.8**	*	0.11	23.8	3.9	*	0.13	29.6	4.1	*	0.13	35.3	**4.9**	*	0.13	37.4	**4.4**	*	0.12
	4	**19.7**	**5.4**			**30.1**	**5.7**			**33.1**	**4.4**			36.6	4.0			39.1	3.8		
	5	**20.4**	3.7			29.3	3.4			**33.1**	2.7			37.6	2.7			41.4	2.9		
	6	17.9	1.8			**29.9**	2.3			**32.8**	1.7			**38.8**	2.0			**44.8**	2.4		
Node Type	BinaryOperation	13.6	5.6			22.1	6.2			26.5	5.1			29.6	4.0			32.2	3.8		
	IfStatement	8.0	-5.6			13.4	-7.0			17.9	-6.6			22.4	-5.2			25.3	-4.5		
	ReturnStatement	7.9	-4.4			16.6	-1.7			22.5	-0.3			26.5	0.0			28.7	-0.4		
	ForStatement	8.1	-2.5			13.8	-2.9			17.3	-3.4			19.4	-4.1			21.4	-4.2		
	MethodInvo.	20.7	**8.8**	*	0.13	28.3	**7.1**	*	0.12	33.5	**6.6**	*	0.12	37.3	**6.2**	*	0.11	39.3	**5.6**	*	0.10
	TernaryExp.	17.5	2.2			21.9	1.0			29.8	1.6			36.0	2.0			**42.1**	2.6		
	StatementExp.	3.5	-1.7			6.9	-2.0			10.3	-2.0			17.2	-1.4			17.2	-1.7		
	WhileStatement	26.3	2.1			**36.8**	1.9			**42.1**	1.8			**42.1**	1.3			**42.1**	1.1		
	VariableDeclare	**28.6**	2.0			28.6	0.9			35.7	1.0			35.7	0.7			35.7	0.5		

Void methods are prone to stubborn mutants (10.6 at Score1 down to 3.3 at Score5), as the absence of return values hampers assertion-based testing. **Mutation operators:** Method-level operators generate more stubborn mutants, over-represented 7.4 at Score1 and still 5.4 at Score4, indicating harder detection. **Nesting depth:** Mutants in deeper code (levels 3–4) are over-represented (5.8→3.9 for level 3, 5.7→3.8 for level 4), reflecting increased structural complexity and reduced control-path observability.

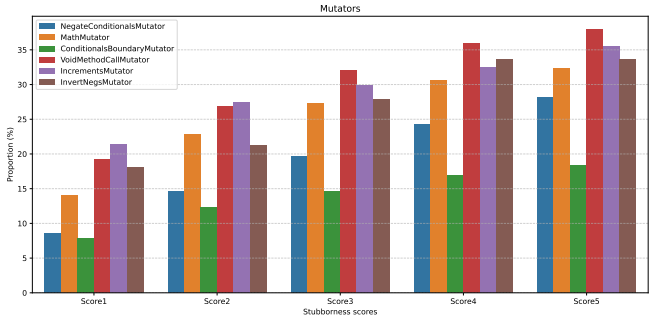
VI. Related Work

The link between equivalent and stubborn mutants was studied by Yao et al. [31], finding certain mutation operator classes, such as LCR, are more likely to produce stubborn mutants. While their focus was on operator classes, we also consider code-level characteristics such as access modifiers, return types, and nesting depth.

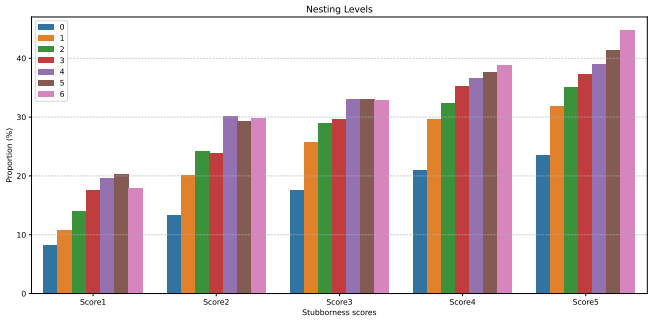
Zhu et al. [33] investigated the relationship between mutation score and code testability and observability, showing that metrics like test directness strongly influence killability and suggesting refactorings to improve scores. Our work complements this by identifying code attributes that correlate with stubbornness across diverse subjects. Papadakis et al. [27] advocated prioritising mutants that correlate with real fault revelation, showing that hard-to-kill mutants may better indicate real faults than subsuming mutants. We build on this by using RSTM to systematically identify stubborn mutants and evaluate their practical utility in test selection and fault reveal rate.



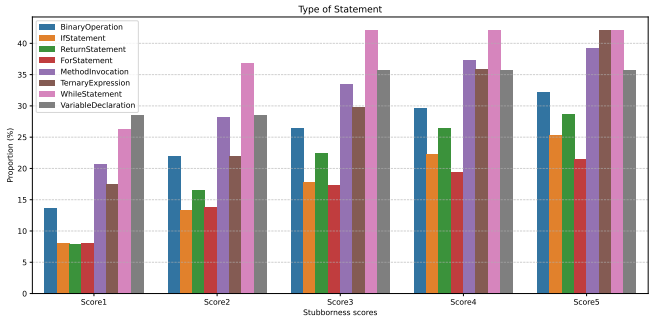
(a) Access Modifiers and Method Types



(b) Mutators



(c) Nesting Levels



(d) Type of Statement

Figure 3: Bar plots showing the distribution of stubborn mutants proportions across different categories and ranks.

Shin et al. [30] proposed a diversity-aware mutation adequacy criterion to improve fault detection, but it does not focus on stubborn mutants. Our work specifically characterises stubborn mutants, quantifies their fault-revealing potential, and evaluates how test selection strategies target them effectively. Chekam et al. [2] used dynamic symbolic execution, Lin et al. [22] applied ranking strategies, and Dang et al. [5] employed co-evolutionary genetic algorithms to detect stubborn mutants more efficiently. These approaches improve test generation by targeting hard-to-kill mutants, whereas we focus on mutants that are already killable but killed by very few tests, evaluating their value for selecting efficient subsets from existing test suites using lightweight, scalable techniques.

Recent ML-based approaches also target mutant selection. Chekam et al. [1] used feature engineering to target stubborn mutants, and Cerebro [14] employs an encoder-decoder to predict subsuming mutants. While effective, these methods rely on prior mutant labels, whereas we evaluate diversity- and coverage-based test selection strategies that are agnostic to stubborn mutants, using RSTM only as a benchmark.

VII. Conclusions and Future Work

This paper presented the first empirical study examining the relationship between test selection strategies and fault difficulty, using stubborn mutants as proxies for difficult faults and also hard-to-find real faults. We compared dynamic coverage-based and lightweight, static diversity-based techniques across seven real-world Java projects. Results show that coverage-based approaches perform well at broader thresholds, but under stricter definitions of stubbornness, bytecode-based FAST outperforms them, requiring up to 46% fewer tests. We also observed strong correlations between mutant stubbornness and code properties, including access modifiers, return types, mutation operator categories, and nesting depth. Tests that exclusively kill stubborn mutants constitute only 2–25% of a suite, yet achieve full mutation adequacy and up to 88% fault-revealing rate.

For future work, we plan to explore stubborn mutant detection using semantic models such as Reachability-Infection-Propagation (RIP) to capture deeper execution characteristics. We will further investigate the relationship between stubborn mutants and testability, examining structural factors (e.g., control-flow constructs) that influence fault-detection difficulty. Methodologically, we aim to refine threshold selection by reducing reliance on S_{max}, which may be sensitive to outliers, and instead consider aggregate statistics such as average killability. We also plan to evaluate weighted APFD to account for fault criticality (e.g., severity or impact) and incorporate cost-aware metrics reflecting test execution time. Finally, we intend to extend experiments to additional systems and languages, and provide qualitative case studies illustrating which tests kill stubborn mutants and why diversity-based approaches are more effective. Combining RSTM with ML-based pipelines may further enable context-sensitive stubbornness measures.

VIII. Acknowledgements

Phil McMinn is supported, in part, by the EPSRC grant “Test FLARE” (EP/X024539/1).

REFERENCES

[1] T. T. Chekam, M. Papadakis, T. F Bissyandé, Y. Le Traon, and K. Sen. Selecting fault revealing mutants. *Empirical Softw. Engg.*, 25(1):434–487, 2020.

[2] T. T. Chekam, M. Papadakis, M. Cordy, and Y. Le Traon. Killing stubborn mutants with symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–23, 2021.

[3] 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 (ISSTA)*, pages 449–452, 2016.

[4] H. Cramér. *Mathematical methods of statistics*, volume 9. Princeton University Press, 1946.

[5] X. Dang, X. Yao, D. Gong, and T. Tian. Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain. *IEEE Transactions on Reliability*, 69(1):334–348, 2020.

[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[7] H. Du, V.K. Palepu, and J.A. Jones. To kill a mutant: An empirical study of mutation testing kills. In *Proceedings of the International Symposium on Software Testing and Analysis (SIGSOFT)*, pages 715–726, 2023.

[8] I. Elgendy, R. Hierons, and P. McMinn. Evaluating string distance metrics for reducing automatically generated test suites. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 171–181, 2024.

[9] I. Elgendy, R. Hierons, and P. McMinn. Empirically evaluating the use of bytecode for diversity-based test case prioritisation. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 171–181, 2025.

[10] I. Elgendy, R. Hierons, and P. McMinn. A systematic mapping study of the metrics, uses and subjects of diversity-based testing techniques. *Software Testing, Verification and Reliability*, 35(2):e1914, 2025.

[11] I. Elgendy, R. Hierons, and P. McMinn. Replication package. https://github.com/islamelgendy/Replication-package-stubborn-mutants, 2026. [Online; accessed 5-Mar-2026].

[12] I. Elgendy, R. Hierons, and P. McMinn. Replication Package for Empirically Evaluating the Use of Bytecode for Diversity-Based Test Case Prioritisation. https://github.com/islamelgendy/Replication-Package-Evaluating-Bytecode-Diversity, 2026. [Online; accessed 5-Mar-2026].

[13] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the SIGSOFT symposium and the European conference on Foundations of software engineering*, pages 416–419, 2011.

[14] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. Le Traon. Cerebro: Static subsuming mutant selection. *IEEE Transactions on Software Engineering*, 49(1):24–43, 2023.

[15] L. Gonzalez-Hernandez, B. Lindström, J. Offutt, S.F. Andler, P. Potena, and M. Bohlin. Using mutant stubbornness to create minimal and prioritized test sets. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, pages 446–457, 2018.

[16] JaCoCo. Jacoco implementation design. http://www.jacoco.org/jacoco/trunk/doc/implementation.html, 2025. [Last accessed: 3-December-2025].

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010.

[18] 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 (ISSTA)*, pages 437–440, 2014.

[19] R. Just, B. Kurtz, and P. Ammann. Inferring mutant utility from program context. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 284–294, 2017.

[20] B. Kurtz, P. Ammann, M. E Delamaro, J. Offutt, and L. Deng. Mutant subsumption graphs. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, pages 176–185, 2014.

[21] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012.

[22] H. Lin, Y. Wang, Y. Gong, and D. Jin. Domain-RIP analysis: A technique for analyzing mutation stubbornness. *IEEE Access*, 7:4006–4023, 2019.

[23] B. Lindström, J. Offutt, L. Gonzalez-Hernandez, and S. F Andler. Identifying useful mutants to test time properties. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 69–76, 2018.

[24] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 222–232, 2018.

[25] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[26] C. Pacheco and M. D Ernst. Randoop: Feedback-directed random testing for java. In *Proceedings of the Companion to the SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA Companion)*, pages 815–816, 2007.

[27] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 32–39, 2018.

[28] K. Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[29] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[30] D. Shin, S. Yoo, and D. Bae. Diversity-aware mutation adequacy criterion for improving fault detection capability. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, pages 122–131, 2016.

[31] 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 (ICSE)*, pages 919–930, 2014.

[32] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[33] Qianqian Zhu, Andy Zaidman, and Annibale Panichella. How to kill them all: An exploratory study on the impact of code observability on mutation testing. *Journal of Systems and Software*, 173:110864, 2021.