Test Flimsiness: Characterizing Flakiness Induced by Mutation to the Code Under Test

Owain Parry
University of Sheffield
Sheffield, UK

Gregory M. Kapfhammer
Allegheny College
Meadville, USA

Michael Hilton
Carnegie Mellon University
Pittsburgh, USA

Phil McMinn
University of Sheffield
Sheffield, UK

Abstract

Flaky tests, which fail non-deterministically against the same version of code, pose a well-established challenge to software developers. In this paper, we characterize the overlooked phenomenon of test **FLIM**siness: **FL**akiness **I**nduced by **M**utations to the code under test. These mutations are generated by the same operators found in standard mutation testing tools. Flimsiness has profound implications for software testing researchers. Previous studies quantified the impact of pre-existing flaky tests on mutation testing, but we reveal that mutations themselves can induce flakiness, exposing a previously neglected threat. This has serious effects beyond mutation testing, calling into question the reliability of any technique that relies on deterministic test outcomes in response to mutations.

On the other hand, flimsiness presents an opportunity to surface potential flakiness that may otherwise remain hidden. Prior work perturbed the execution environment to augment rerunning-based detection and the test code to support benchmarking. We advance these efforts by perturbing a third major source of flakiness: the code under test. We conducted an empirical study on over half a million test suite executions across 28 Python projects. Our statistical analysis on over 30 million mutant-test pairs unveiled flimsiness in 54% of projects. We found that extending the standard rerunning flaky test detection strategy with code-under-test mutations detects a substantially larger number of flaky tests (median 740 vs. 163) and uncovers many that the standard strategy is unlikely to detect.

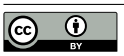
CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Software Testing, Mutation Testing, Flaky Tests.

ACM Reference Format:
Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2026. Test Flimsiness: Characterizing Flakiness Induced by Mutation to the Code Under Test. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil* . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773125



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773125

1 Introduction

Flaky tests, which are test cases that fail non-deterministically against the same version of code [37], represent a persistent challenge in software development [18, 22, 42]. This issue is alarmingly widespread, with survey data revealing that 59% of software developers grapple with flaky tests on a monthly, weekly, or even daily basis [39]. For developers, the challenge of flaky tests is profound: they erode trust in test suites, lead to wasted time debugging spurious failures that are not indicative of genuine bugs, and cause false alarms that hold up continuous integration pipelines [17, 20, 25]. Flaky tests also pose challenges for researchers, including the risk of invalidating determinism assumptions made by automated techniques. The limitations of standard detection approaches and existing datasets also hinder the benchmarking of mitigation strategies and the assessment of the wider impact of flaky tests.

Challenge 1: The Hidden Hazard of Mutation-Induced Flakiness. Flaky tests threaten the validity of techniques that assume deterministic test behavior, including fault localization [53], automatic test suite generation [23], and mutation testing. Prior work [8, 48] showed how flaky tests compromise mutation testing, but their focus was limited to pre-existing flakiness, leaving open the question of whether new flakiness might arise during analysis. This leaves a critical gap: if mutation operators applied to the code under test can trigger flakiness in stable tests, then mutation testing may be subject to unaccounted-for sources of uncertainty. This concern is not limited to mutation testing. Many techniques driven by mutants, including within the fields of fault localization [36, 55, 56], regression testing [19], performance testing [45], and web testing [32], assume deterministic test outcomes. If standard mutation operators can induce flakiness, then the reliability of conclusions drawn from mutant-driven techniques may be compromised—a possibility that, despite its risk, remains largely unexplored.

Challenge 2: Inadequate Detection, Incomplete Datasets. At the same time, detecting flaky tests remains a fundamental issue. Rerunning test suites to observe inconsistent failures is a popular strategy [7, 21, 40, 41], but often fails to expose flaky tests that fail only under specific conditions. To address this, prior work has altered the execution environment to increase the likelihood of failures, such as by scheduling competing stressor tasks [51] or restricting resource allocations [50]. Another problem is the scarcity and poor reproducibility of real-world flaky test datasets, which hampers the evaluation of mitigation strategies and the study of the broader impact of flakiness on techniques including mutation testing. To overcome this, researchers have injected artificial flakiness

```
    def parse_chunked(self, unreader):
        (size, rest) = self.parse_chunk_size(unreader)
        while size > 0:
-           while size > len(rest):
+           while size < len(rest):
                size -= len(rest)
                yield rest
                rest = unreader.read()
                if not rest:
                    raise NoMoreData()
```

(a) An excerpt from the mutant diff. Lines prefixed with - indicate removed code, while those prefixed with + indicate added code.

```
@pytest.mark.parametrize("fname", httpfiles)
def test_http_parser(fname):
    env = treq.load_py(os.path.splitext(fname)[0] + ".py")
    expect = env["request"]
    cfg = env["cfg"]
    req = treq.badrequest(fname)
    with pytest.raises(expect):
        req.check(cfg)
```

(b) The test case source code.

Figure 1: An example of a flimsy mutant-test pair from the gunicorn project. Based on our own inspection of the code, we reasoned that changing the loop condition to `while size < len(rest)` causes the parser to consume data even when it already has enough, subtracting too much and potentially sending `size` negative. This would misalign chunk boundaries and could raise `NoMoreData`. Assuming that the amount of data returned by `read()` varies depending on factors like buffering or timing, this change exposes the test case to input/output-related flakiness, an established category in the literature [18, 33, 37].

into test code, either by instrumenting bytecode to probabilistically raise exceptions [15], or by applying custom mutation operators that introduce flaky anti-patterns [13]. While existing work has explored perturbing the environment and test code, flakiness has been shown to arise from a third source: the code under test [18, 22, 33]. This opens up a complementary opportunity: rather than viewing mutation as a source of noise, we might treat it as a lens that reveals potential flakiness that conventional rerunning fails to detect.

We set out to explore these risks and opportunities by characterizing a previously overlooked phenomenon that we refer to as *test flimsiness*, defined as **FL**akiness **I**nduced by **M**utations to the code under test (**FLIM**siness). Unlike prior work that used specialized mutation operators targeting test code [13], these mutations are automatically generated by the same standard operators found in out-of-the-box mutation testing tools [29, 43]. We conducted a large-scale empirical study across 28 Python projects, comprising over half a million test suite executions. These projects span diverse domains from web development and data science to networking, cryptography, and scientific computing. Using robust statistical analysis on more than 30 million mutant-test pairs (a mutation and a test case, evaluated together to detect behavioral changes), we reliably identified transitions from stability to flakiness triggered by mutations to the code under test. See Figure 1 for a concrete example of a flimsy mutant-test pair from the gunicorn project, one of the 28 Python projects used in this paper's empirical study. Ultimately, we found flimsiness to be prevalent, affecting 15 out of 28 projects, highlighting the extent to which mutant-driven techniques can induce flakiness and undermine their own reliability.

We investigated whether specific mutation operators are more likely to induce flakiness and found that the mutants from one particular operator have over 3 times higher odds of inducing flakiness compared to others. This suggests that selecting mutation operators more strategically could improve the efficiency of approaches that use mutants to expose flakiness. We applied a simulation-based methodology to compare the standard rerunning flaky test detection strategy to an alternative mutation-based strategy. We found that the mutation-based approach detects a substantially larger number of flaky tests (median 740 vs. 163) and uncovers many that the standard strategy is unlikely to detect. Although we do not assert that this induced flakiness fully reflects naturally occurring flakiness, these results emphasize the potential of mutations to the code under test as an augmentation to rerunning, and reinforces our interpretation of flimsiness as a distinct form of flakiness. Finally, we evaluated the predictive value of pre-mutation coverage of the mutated line with respect to predicting the occurrence of flimsiness. In particular, we found that whether a test case non-deterministically covered the mutated source code line prior to mutation does not reliably predict whether the mutant will induce flakiness (precision: 0.16, recall: 0.1). This suggests that the causes of flimsiness are more nuanced than simply surfacing unstable coverage, despite previous studies that highlight the issue [27, 48].

Taken together, our findings highlight a previously understudied threat to mutant-driven techniques while also revealing a new avenue for surfacing flaky behavior through code-under-test mutation. In summary, this paper makes the following key contributions:

1. Definition and characterization of test flimsiness. We address a crucial gap in flakiness research by empirically analyzing test flimsiness, a phenomenon in which stable test cases become flaky in response to mutations applied to the code under test.

2. Extensive empirical evaluation. Our large-scale study, conducted across a diverse set of 28 real-world Python projects, reveals that test flimsiness occurs in 54% of projects, among other timely insights into this heretofore overlooked phenomenon.

3. Comprehensive public dataset. We release a rich dataset [5] comprising over half a million test suite runs and 30 million mutant-test pairs, annotated with test outcome reports, coverage data, and other metadata to support future research in this area.

2 Methodology

This section describes our methodology for answering this paper's four research questions regarding test flimsiness:

RQ1: Prevalence. How prevalent is mutation-induced flakiness among otherwise stable test cases?

RQ2: Operators. Which mutation operators are the most likely to produce mutants that induce test flakiness?

RQ3: Detection. To what extent do test suite runs with and without mutations detect the same flaky tests?

RQ4: Coverage. How well does pre-mutation coverage of the mutated line predict whether the mutant induces test flakiness?

We developed Python scripts and a custom pytest plugin to automate the empirical study. These are available in the replication package [5] alongside our dataset and evaluation results.

2.1 Dataset

We selected 30 active Python projects to use as subjects in our study of test flimsiness. These projects are all hosted on GitHub and appear on a larger list of projects that the Open Source Security Foundation recognizes as influential and important [4]. This generated list ranks repositories according to metrics such as contributor activity, project maturity, release frequency, and dependency usage, highlighting those that are most essential to the open-source ecosystem. We also selected some projects because prior flaky test studies used them as subjects [38, 40, 41], and we randomly sampled others from the influential projects list. For each candidate subject, we spent up to 30 minutes attempting to set it up and run its test suite. We used the commit associated with the latest release or tag when available, and otherwise the latest commit on the main branch. If we could not complete setup within this time limit, we excluded the project. For this study, we dropped two projects due to persistent technical issues, including frequent crashes and timeouts that disrupted the experiment pipeline. Following this process ultimately yielded a final dataset of 28 diverse subjects.

For each subject, our scripts executed the test suite 20 times with line coverage measurement, referred to as the *coverage runs*, and 12,000 times without, referred to as the *standard runs*. We used Coverage.py [2], a mature and widely used Python coverage tool, and saved coverage data as SQLite3 databases. Our scripts generated 300 mutated versions of each subject by applying a randomly selected mutation operator to a randomly selected line in the code under test, entirely excluding test code, using Cosmic Ray [1]. To ensure test code (including the source files of tests and support code such as fixture definitions) was excluded from mutation, we defined filename patterns using wildcards for each subject. We chose Cosmic Ray for its active maintenance and modular design suitable for our study. We excluded two of Cosmic Ray's mutation operators (ExceptionReplacer and RemoveDecorator) due to persistent issues, including frequent generation of unexecutable mutants. For the first 100 mutants of each subject, our scripts executed the test suite 100 times, and for the remaining 200, our scripts executed the test suite 10 times; all such runs are collectively referred to as the *mutant runs*. Since we could not know in advance how many runs per mutant would be appropriate, we followed this approach to strike a balance between exploration and exploitation. The 12,000 mutant runs ($100 \times 100 + 200 \times 10$) matches the number of standard runs.

For each test suite run, our scripts used a Docker [3] container and saved the test report in JUnit XML format. Docker containers ensure a fresh filesystem for each test suite run, among other isolation features, mitigating any potential dependencies between runs. JUnit XML is a standard format for the results of test suite runs that includes test case outcomes, tracebacks, and any captured output. We scheduled standard and mutant runs concurrently and randomized their order to mitigate infrastructure flakiness [22], ensuring any such noise affected both run types evenly and prevented spurious conclusions about mutant effects. Finally, our scripts recorded detailed metadata for each mutant, including the mutation operator, the target line, and diffs with extended context (using `git diff -unified=15`). Our minimal pytest plugin automatically collects coverage data, resource metrics, and test source code during execution without interfering with test outcomes or performance.

2.2 Methodology for RQ1: Prevalence

For each mutant-test pair (each combination of mutant and test case within a subject), we set out to evaluate the composite null hypothesis: "no difference in failure rate between standard and mutant runs *or* the flimsiness condition C is false". Where f_s is the number of standard runs where the test case failed, f_m is the number of mutant runs where it failed, and r_m is the total number of mutant runs (either 10 or 100), the flimsiness condition, $C \equiv (f_s = 0) \wedge (0 < f_m < r_m)$, defines *possible* cases of mutation-induced flakiness. Including the flimsiness condition here enables us to focus on relevant pairs, as opposed to all pairs where there was a significant change in failure rate (e.g., when a test case kills a mutant). It avoids circular reasoning because the condition is prespecified using raw failure counts and is independent of p-values.

Our scripts calculated p-values for each pair-level hypothesis. For pairs where C is true, our scripts calculated the p-value using a two-tailed Boschloo's exact test [12]. We selected this statistical test because it is optimal for highly unbalanced contingency tables (12,000 standard vs. 10/100 mutant runs) and is more powerful than Fisher's exact test [34]. For pairs where C is false, our scripts set the p-value to 1. When C is false, the composite null is automatically true, and setting the p-value to 1 is a conservative approach. Our scripts adjusted the raw p-values for each pair by applying the Benjamini-Yekutieli (BY) correction [10] to control the false discovery rate (FDR) at the pair level. Significant pairs (adjusted p-value $\leq \alpha$) provide FDR-controlled evidence that the mutant caused a change in failure rate *and* the test case transitioned from stable to flaky. This implies that the mutant induced flakiness in the test case. We selected the BY correction because it is valid under arbitrary dependence. In the context of our study, such dependence could arise from systemic flakiness [42].

For each mutant, we evaluated the global null hypothesis: "for all pairs associated with this mutant the pair-level composite null hypothesis is true". To do so, our scripts calculated family p-values for each mutant-level hypothesis using Bonferroni correction [11] over the raw individual p-values for each associated pair (before BY correction). We selected Bonferroni correction because it is valid under arbitrary dependence. At this level, dependence could arise from the fact that the mutations were applied to the same codebase. Our scripts adjusted the raw family p-values for each mutant using BY correction to control the FDR at the mutant level. Significant mutants (adjusted family p-value $\leq \alpha$) provide evidence that the mutant induced flakiness in at least one test case.

For each test case, we evaluated the global null hypothesis: "for all pairs associated with this test case the pair-level composite null hypothesis is true". Our scripts adhered to the same procedure of applying Bonferroni correction to produce the raw family p-values followed by the BY correction to control the FDR at the test case level. Significant test cases provide evidence that at least one mutant induced flakiness in the test case. For each subject and in total, our scripts counted the number of significant pairs, mutants, and test cases, setting the significance threshold at $\alpha = 0.01$.

2.3 Methodology for RQ2: Operators

Our goal was to identify specific mutation operators that are significantly more or less likely than others to produce mutants that induce test flakiness. For each mutation operator that we used in our study, the scripts counted the number of significant and non-significant mutants (as per our RQ1 methodology) produced by that operator across all subjects. From these counts, they created a 2×2 contingency table for each operator comparing these counts to the corresponding counts for all the other operators combined. For each table, our scripts used the two-tailed Boschloo's test to calculate a p-value for the null hypothesis that the corresponding operator is as likely to produce mutants that induce flakiness as any other operator. They also applied the BY correction to these p-values to control the FDR and calculated the natural logarithm of the odds ratio with Haldane-Anscombe correction [31] to quantify the magnitude and direction of any effect. The odds ratio itself quantifies how many times greater or smaller the odds (the ratio of the probability of an event occurring to that of it not occurring) are that mutants from a specific operator induce test flakiness versus others. We used the logarithm because it produces a more interpretable effect size centered around zero, where negative values indicate that an operator is less likely to produce such mutants and positive values indicate the opposite. We used the Haldane-Anscombe correction, which involves adding 0.5 to each cell, to avoid invalid values.

We repeated this analysis with respect to the number of mutants produced by each operator that are deterministically killed by at least one test case. For a given mutant-test pair, the mutant is killed by the test case if and only if $(f_s = 0) \wedge (f_m = r_m)$. We compared the effect sizes for both criteria to establish if any operator effects are unique to inducing flakiness or just reflective of general disruption to the code under test. To do so, our scripts performed Spearman's rank-order correlation analysis [52] between the two sets of effect sizes, producing a p-value for the null hypothesis of no correlation, and a correlation coefficient ρ. The coefficient ranges from -1 to 1, indicating perfect negative or positive correlations respectively. We used Spearman's rank-order because it captures monotonic relationships without assuming linearity or normality [47].

2.4 Methodology for RQ3: Detection

We aimed to assess how the standard flaky test detection strategy of repeatedly executing the test suite (without mutations applied to the code under test) to observe inconsistent failures differs from an alternative mutation-based strategy. The mutation-based rerunning strategy is to randomly apply mutation operators to the code under test and repeatedly execute the test suite against each mutant to observe inconsistent failures with respect to the same mutant (not between mutants). Our approach was to compare the *agreement* of both strategies in terms of the number of flaky tests they detect in common to the *consistency* of both strategies in terms of the number of flaky tests each detects reliably across repeated invocations. This enables us to establish if a low agreement is genuinely reflective of a fundamental difference between the two strategies or is simply an artifact of low consistency (high variability) within either strategy.

For each subject, our scripts sampled 1,000 times: a random subset of half the standard runs and, for each mutant, a random subset of half the mutant runs. For each sample i, our scripts constructed two sets of detected flaky tests: the *standard-detected* set S_i, representing the flaky tests detected by the standard rerunning strategy, and the *mutation-detected* set M_i, representing the flaky tests detected by the mutation-based strategy. Our scripts assigned a test case to S_i if and only if $0 < f'_s < r'_s$, where f'_s and r'_s are the number of failed and total sampled standard runs respectively. Our scripts assigned a test case to M_i if and only if there exists at least one mutant such that $0 < f'_m < r'_m$, where f'_m and r'_m are the number of failed and total sampled mutant runs respectively. Over the 1,000 samples, our scripts produced empirical distributions for $|S_i|$, $|M_i|$ and $|S_i \cap M_i|$ by computing the size of both sets and the size of their intersection within each sample i. The scripts also produced distributions for $|S_i \cap S_j|$ and $|M_i \cap M_j|$ by computing intersection sizes between pairs of standard-detected and mutation-detected sets across independent samples i and j.

For each subject, our scripts compared the distribution for $|S_i \cap M_i|$, which measures the agreement between the two detection strategies, to the distributions for $|S_i \cap S_j|$ and $|M_i \cap M_j|$, which measure the consistency of both strategies respectively. To make both comparisons, our scripts used the two-tailed Mann-Whitney U test [35]. The null hypothesis is that the distribution for $|S_i \cap M_i|$ comes from the same population as the distribution for either $|S_i \cap S_j|$ or $|M_i \cap M_j|$, depending on which comparison is being performed. We selected the Mann-Whitney U test because it is a non-parametric method that compares two distributions without assuming normality [54]. Our scripts applied BY correction to the resultant p-values to control the FDR and also calculated the rank biserial correlation (RBC) to provide interpretable, direction-sensitive effect sizes. For either comparison, a negative RBC would indicate that the agreement tends to be fewer than the consistency, and a positive RBC would indicate the opposite.

To perform an aggregate-level analysis, we repeated the same procedure, but instead of analyzing each subject separately, we merged their results. For each sample, our scripts took the union of all standard-detected flaky test sets across subjects, and likewise for the mutation-detected sets. This produced one global standard set and one global mutant set per sample, over which our scripts repeated the same agreement and consistency analysis.

2.5 Methodology for RQ4: Coverage

We sought to evaluate the extent to which test flimsiness can be explained as a manifestation of unstable line coverage, an issue highlighted by prior studies [27, 48]. That is, whether mutating a line that was previously executed non-deterministically causes the test case to fail only when that line happens to be executed post-mutation, transforming inconsistent coverage into visible flakiness. For each mutant-test pair, our scripts recorded how many times the test case passed versus how many it covered the would-be mutated line during the passing runs, using the results of the subject's 20 coverage runs (pre-mutation). Using this, our scripts computed two binary indicators for each pair. The first indicates that the mutated line was *flakily covered*, and is true for a given pair if and only if the test case covered the mutated line at least once during the passing runs but fewer than the total number of passing runs. Conditioning on passing runs is more conservative because it avoids spuriously labelling a line as flakily covered when a test case failed due to

Table 1: For each of the 28 subjects, the counts of mutant-test pairs, mutants, and test cases. These counts are broken down into overall counts, those that are possible evidence of flimsiness, and those that are significant (Sig.) evidence.

	# Pairs			# Mutants			# Tests		
		Possible			Possible			Possible	
Subject Name			**Sig.**			**Sig.**			**Sig.**
PyCQA/bandit	81900	0	0	300	0	0	273	0	0
mozilla/bleach	134400	0	0	300	0	0	448	0	0
quantumlib/Cirq	5339400	475	359	300	8	5	17798	272	210
cylc/cylc-flow	814500	3	1	300	3	1	2715	3	1
dask/dask	3292500	17	5	300	1	1	10975	17	5
spesmilo/electrum	206400	0	0	300	0	0	688	0	0
eventlet/eventlet	186300	0	0	300	0	0	621	0	0
falconry/falcon	1098900	1	0	300	1	0	3663	1	0
pallets/flask	147300	0	0	300	0	0	491	0	0
benoitc/gunicorn	78000	7	4	300	4	3	260	5	3
ipython/ipython	285000	3	2	300	3	2	950	2	1
apache/libcloud	2454000	7	0	300	7	0	8180	1	0
Delgan/loguru	465300	11	10	300	2	2	1551	11	10
mitmproxy/mitmproxy	546000	25	8	300	9	4	1820	21	8
more-itertools/more-itertools	200700	0	0	300	0	0	669	0	0
networkx/networkx	1412400	5	5	300	3	3	4708	5	5
nltk/nltk	133800	2	0	300	2	0	446	1	0
oauthlib/oauthlib	204000	17	16	300	1	1	680	17	16
PrefectHQ/prefect	3534300	232	194	300	24	12	11781	191	162
PyGithub/PyGithub	298500	0	0	300	0	0	995	0	0
pyparsing/pyparsing	561000	0	0	300	0	0	1870	0	0
psf/requests	177000	1	1	300	1	1	590	1	1
saltstack/salt	3759000	79	0	300	6	0	12530	79	0
encode/starlette	267900	4	0	300	4	0	893	2	0
vertexproject/synapse	342600	27	26	300	1	1	1142	27	26
twisted/twisted	2866800	45	31	300	18	16	9556	43	31
urllib3/urllib3	515700	29	17	300	11	4	1719	26	17
xonsh/xonsh	1533600	55	30	300	7	2	5112	53	30
Total	30937200	1045	709	8400	116	58	103124	778	526

flakiness before reaching that line. The second indicates that the mutated line was *simply covered*, and is true for a given pair if and only if the test case covered the mutated line at least once during passing runs (flakily covered implies simply covered).

The experiment scripts evaluated how well these two indicators predict whether a mutant-test pair exhibits statistically significant evidence of flimsiness, as defined in RQ1. To do so, our scripts calculated the confusion matrix, precision, and recall for both indicators at the level of each subject, and at the aggregate level by summing the confusion matrix elements across all subjects. In this context, a mutant-test pair is a true positive (TP) if the indicator correctly predicts that the pair is significant, a false positive (FP) if it predicts the pair as significant when it is not, a false negative (FN) if it misses a significant pair, and a true negative (TN) if it correctly predicts a non-significant pair. Precision is the proportion of predicted significant pairs that are truly significant, written as $TP \div (TP + FP)$, while recall is the proportion of actual significant pairs that were correctly predicted, defined as $TP \div (TP + FN)$. We used the simply-covered indicator as a point of comparison to verify that any predictive power attributed to being flakily covered is not solely due to the mutated source code line being covered.

2.6 Threats to Validity

Although our methodology's design supports the rigorous investigation of test flimsiness, we acknowledge several validity threats. The generalizability of our findings is inherently limited by our dataset. Although we mitigated this by selecting a broad range of open-source Python projects from diverse domains, the results may not reflect the nature of test flimsiness in other programming languages. It is possible that the 12,000 standard runs per subject were insufficient to manifest all pre-mutation flaky tests, particularly those with very low failure rates, and similarly for the 10 or 100 mutant runs. Likewise, the 20 coverage runs performed per subject may not have captured all instances of non-deterministic coverage. As previously highlighted [9, 26], this is a general threat to any empirical study of flaky tests that cannot be fully eliminated,

Table 2: For each mutation operator: the number of mutants that induce flakiness in at least one test case (Pos.), those that do not (Neg.), the log odds ratio (LOR) of the contingency table, the p-value (P-Val.) of the corresponding null hypothesis, and the BY-adjusted p-value (Q-Val.). Similarly for being killed by at least one mutant. LOR correlation analysis results: $\rho = 0.66$, $p = 0.02$.

	Inducing Flakiness					Being Killed				
Operator Name	**Pos.**	**Neg.**	**LOR**	**P-Val.**	**Q-Val.**	**Pos.**	**Neg.**	**LOR**	**P-Val.**	**Q-Val.**
AddNot	9	823	0.56	0.188	0.266	535	297	0.81	<0.001	<0.001
NumberReplacer	5	1186	-0.48	0.249	0.315	434	757	-0.48	<0.001	<0.001
ReplaceAndWithOr	3	112	1.53	0.070	0.112	69	46	0.55	0.004	0.013
ReplaceBinaryOperator	10	3377	-1.14	<0.001	0.001	1450	1937	-0.25	<0.001	<0.001
ReplaceBreakWithContinue	0	15	1.52	>0.999	>0.999	6	9	-0.24	0.797	0.869
ReplaceComparisonOperator	18	2014	0.36	0.237	0.315	984	1048	0.10	0.042	0.072
ReplaceContinueWithBreak	0	21	1.20	>0.999	>0.999	6	15	-0.73	0.144	0.216
ReplaceFalseWithTrue	0	149	-0.76	0.620	0.744	53	96	-0.46	0.008	0.019
ReplaceOrWithAnd	3	83	1.83	0.032	0.059	58	28	0.87	<0.001	<0.001
ReplaceTrueWithFalse	0	139	-0.69	0.669	0.765	48	91	-0.50	0.005	0.013
ReplaceUnaryOperator	6	280	1.27	0.020	0.044	168	118	0.51	<0.001	<0.001
ZeroIterationForLoop	4	143	1.55	0.028	0.055	92	55	0.66	<0.001	<0.001

but rather mitigated by performing as many test suite runs as resources permit. Performing over 30 million hypothesis tests raises the risk of false discoveries, even with the BY control. To mitigate this concern, we used a strict significance threshold of $\alpha = 0.01$.

When analyzing operator effects, the Haldane-Anscombe correction may bias small-sample effect sizes, though it is necessary to avoid infinite or undefined values. We mitigated this by supplementing odds ratios with Boschloo p-values for greater robustness. Defects in our scripts could have impacted the evaluation and led to incorrect conclusions. To reduce this risk, we relied on well-established, open-source Python libraries, such as SciPy [6], for all critical statistical analyses. Given their wide adoption and active maintenance, we are confident that any library-level bugs would be quickly identified, documented, and patched. Our requirement that the baseline failure rate be zero ($f_s = 0$) as part of the flimsiness condition excludes tests with pre-existing flakiness, potentially omitting mutant-test pairs where the mutation exacerbates rather than introduces flakiness. While this remains an interesting direction for future work, we deliberately scoped our study to newly induced flakiness (stable-to-flaky transitions) to preserve causal clarity. Infrastructure flakiness [22], such as transient network instability, may have introduced bursts of flakiness across concurrently executing test suite runs. Such occurrences could be misattributed to flimsiness. We mitigated this through randomization of the test suite run order, interleaving standard and mutant runs, and the use of Docker-based isolation to reduce environmental variability.

3 Results

3.1 Results for RQ1: Prevalence

For mutant-test pairs, mutants, and test cases, Table 1 gives the overall count, the count that are possible evidence of test flimsiness, and the count that are significant evidence ($\alpha = 0.01$), for each subject and in total. A pair is possible evidence if the flimsiness condition C is true. A mutant is possible evidence if C is true for any of its associated pairs, similarly for test cases. Because the pair-level composite null hypothesis includes C, significant evidence at any level implies possible evidence.

Test flimsiness is a prevalent phenomenon. It exists in 15 out of 28 subjects (54%). While most significant pairs and test cases are concentrated among a few subjects (e.g., Cirq and prefect), the significant mutants are distributed more evenly. This implies that a single mutant may induce flakiness in many test cases. In total, 0.7% of mutants induced flakiness in at least one test case.

3.2 Results for RQ2: Operators

For both criteria (inducing flakiness in, or being killed by, at least one test case), Table 2 presents the number of mutants that meet the condition (positive examples) and those that do not (negative examples), as produced by each mutation operator. In addition, the table reports the log odds ratio (LOR) effect size and the p-value for the null hypothesis that the operator is as likely as any other operator to produce mutants that meet the condition.

Certain mutation operators are significantly more likely to produce mutants that induce test flakiness than others. One such example is ReplaceUnaryOperator, with a small adjusted p-value of 0.044, we can reject this null hypothesis at the 5% significance level. The LOR of 1.27 indicates that mutants from this operator have over 3 times higher odds of inducing flakiness compared to others ($e^{1.27} \approx 3.56$). Conversely, ReplaceBinaryOperator has a small p-value but a LOR of -1.14, meaning the odds that a mutant induces flakiness are about a third of that of other operators.

Operators that produce mutants that are easier for test cases to kill are often also good at inducing flakiness. This comes from the results of the Spearman's rank-order correlation analysis between the LOR effect sizes of both criteria. The p-value for the null hypothesis of no correlation is 0.02, indicating that we can reject it at the 2% significance level and conclude that there is a correlation. The coefficient ρ is 0.66, indicating a moderately strong positive relationship. This means that the LOR rankings across operators are generally aligned: operators ranked higher for inducing flakiness tend to be ranked higher for being killed.

Table 3: For each subject and at the aggregate level: the median number of flaky tests detected by the standard and mutation-based strategies; the median number detected in common by both strategies (agreement) and by two independent invocations of each strategy (standard and mutation consistency); and the RBC effect size, p-value (P-Val.), and BY-adjusted p-value (Q-Val.), from the comparison of the agreement distribution to each consistency distribution. Excludes subjects without flaky tests.

	Median					$\lvert S_i \cap M_i \rvert$ v $\lvert S_i \cap S_j \rvert$			$\lvert S_i \cap M_i \rvert$ v $\lvert M_i \cap M_j \rvert$		
Subject Name	$\lvert S_i \rvert$	$\lvert M_i \rvert$	$\lvert S_i \cap M_i \rvert$	$\lvert S_i \cap S_j \rvert$	$\lvert M_i \cap M_j \rvert$	**RBC**	**P-Val.**	**Q-Val.**	**RBC**	**P-Val.**	**Q-Val.**
quantumlib/Cirq	1	247	1	1	232	0.06	0.002	0.003	-1.00	<0.001	<0.001
cylc/cylc-flow	2	4	2	2	3	-0.23	<0.001	<0.001	-0.98	<0.001	<0.001
dask/dask	1	13	0	0	9	-0.04	0.079	0.099	-1.00	<0.001	<0.001
eventlet/eventlet	4	4	2	3	3	-0.28	<0.001	<0.001	-0.32	<0.001	<0.001
falconry/falcon	0	1	0	0	1	0.00	>0.999	>0.999	-0.55	<0.001	<0.001
benoitc/gunicorn	0	4	0	0	3	0.00	>0.999	>0.999	-1.00	<0.001	<0.001
ipython/ipython	0	2	0	0	1	0.00	>0.999	>0.999	-1.00	<0.001	<0.001
apache/libcloud	4	5	0	4	1	-0.32	<0.001	<0.001	-0.32	<0.001	<0.001
Delgan/loguru	0	8	0	0	7	0.00	>0.999	>0.999	-1.00	<0.001	<0.001
mitmproxy/mitmproxy	1	16	1	1	13	0.11	<0.001	<0.001	-1.00	<0.001	<0.001
networkx/networkx	0	5	0	0	4	0.00	>0.999	>0.999	-1.00	<0.001	<0.001
nltk/nltk	0	2	0	0	2	0.14	<0.001	<0.001	-0.41	<0.001	<0.001
oauthlib/oauthlib	0	16	0	0	16	0.00	>0.999	>0.999	-1.00	<0.001	<0.001
PrefectHQ/prefect	43	205	31	25	175	-0.04	0.099	0.121	-1.00	<0.001	<0.001
psf/requests	0	1	0	0	1	0.00	>0.999	>0.999	-0.82	<0.001	<0.001
saltstack/salt	13	16	7	8	9	-0.55	<0.001	<0.001	-0.63	<0.001	<0.001
encode/starlette	1	2	0	1	1	-0.53	<0.001	<0.001	-0.79	<0.001	<0.001
vertexproject/synapse	8	28	4	7	26	-0.94	<0.001	<0.001	-1.00	<0.001	<0.001
twisted/twisted	5	38	3	5	33	-0.57	<0.001	<0.001	-1.00	<0.001	<0.001
urllib3/urllib3	10	28	6	8	24	-0.91	<0.001	<0.001	-1.00	<0.001	<0.001
xonsh/xonsh	30	75	28	28	56	0.02	0.333	0.396	-1.00	<0.001	<0.001
Aggregate	163	740	86	106	637	-0.64	<0.001	<0.001	-1.00	<0.001	<0.001

3.3 Results for RQ3: Detection

For each subject and at the aggregate level, Table 3 presents the median number of flaky tests detected by the standard rerunning strategy and the mutation-based rerunning strategy. It also reports the median number of flaky tests detected in common by both strategies (median agreement), by two independent invocations of the standard strategy (median standard consistency), and by two independent invocations of the mutation-based strategy (median mutation consistency). It also gives the RBC effect sizes and the adjusted p-values from the comparisons of the agreement distribution to the consistency distributions of both strategies.

The two detection strategies detect largely different sets of flaky tests. This is supported by the small adjusted p-values from both comparisons at the aggregate level, indicating that we can confidently reject the corresponding null hypotheses, and the fact that both RBC effect sizes are negative, indicating that the agreement tends to be lower than the consistency for both strategies.

The flaky tests detected by the mutation-based strategy are especially unlikely to be detected by the standard strategy. Notably, the RBC effect size is -1 when comparing agreement to mutation consistency, but only -0.64 when comparing to standard consistency. This reinforces the interpretation of test flimsiness as a distinct manifestation of flakiness that standard rerunning, without mutations to the code under test, is unlikely to detect.

3.4 Results for RQ4: Coverage

For both coverage indicators, Table 4 presents the confusion matrix, precision, and recall for each subject and at the aggregate level.

Whether a test case non-deterministically covered the mutated line prior to mutation does not reliably predict whether the mutant will induce flakiness. This can be inferred from the low precision and recall scores for the flakily covered indicator, for most subjects and at the aggregate level (0.16 and 0.10, respectively). This implies that the causes of test flimsiness are more nuanced than simply manifesting unstable coverage.

Whether a test case covered the mutated line at least once, even if inconsistently, is entirely uninformative as a predictor of flimsiness. Despite a high recall score of 0.8 at the aggregate level, the precision score for the simply covered indicator is 0, rendering it totally ineffective, particularly when compared to the flakily covered indicator, which, although still weak, offers a more balanced precision-recall trade-off. The high recall aligns with the intuition that a test case must exercise the mutated line to be affected. However, since it is less than 1, this result shows that a mutant inducing flakiness does not always imply that the test case covered the mutated line of source code beforehand.

Table 4: For each subject and at the aggregate level: the confusion matrix elements (TP, FP, FN, TN), precision (Pr.), and recall (Re.) for predicting whether a mutant will induce flakiness in a test case based on whether the test case non-deterministically covered the mutated line prior to mutation (flakily covered). Similarly for whether the test case covered the mutated line at least once, even if inconsistently (simply covered). Dashes indicate undefined values resulting from a division by zero.

	Flakily Covered						Simply Covered					
Subject Name	**TP**	**FP**	**FN**	**TN**	**Pr.**	**Re.**	**TP**	**FP**	**FN**	**TN**	**Pr.**	**Re.**
PyCQA/bandit	0	0	0	81900	-	-	0	5593	0	76307	0.00	-
mozilla/bleach	0	0	0	134400	-	-	0	9934	0	124466	0.00	-
quantumlib/Cirq	44	16	315	5339025	0.73	0.12	359	50878	0	5288163	0.01	1.00
cylc/cylc-flow	0	1	1	814498	0.00	0.00	1	12783	0	801716	0.00	1.00
dask/dask	0	261	5	3292234	0.00	0.00	5	52626	0	3239869	0.00	1.00
spesmilo/electrum	0	1	0	206399	0.00	-	0	2805	0	203595	0.00	-
eventlet/eventlet	0	30	0	186270	0.00	-	0	3753	0	182547	0.00	-
falconry/falcon	0	0	0	1098900	-	-	0	7180	0	1091720	0.00	-
pallets/flask	0	0	0	147300	-	-	0	6346	0	140954	0.00	-
benoitc/gunicorn	0	9	4	77987	0.00	0.00	4	5139	0	72857	0.00	1.00
ipython/ipython	0	0	2	284998	-	0.00	2	3129	0	281869	0.00	1.00
apache/libcloud	0	0	0	2454000	-	-	0	12687	0	2441313	0.00	-
Delgan/loguru	0	0	10	465290	-	0.00	9	23796	1	441494	0.00	0.90
mitmproxy/mitmproxy	1	11	7	545981	0.08	0.12	1	4942	7	541050	0.00	0.12
more-itertools/more-itertools	0	0	0	200700	-	-	0	1166	0	199534	0.00	-
networkx/networkx	1	0	4	1412395	1.00	0.20	5	2941	0	1409454	0.00	1.00
nltk/nltk	0	0	0	133800	-	-	0	187	0	133613	0.00	-
oauthlib/oauthlib	0	0	16	203984	-	0.00	16	7947	0	196037	0.00	1.00
PrefectHQ/prefect	21	11	173	3534095	0.66	0.11	147	39908	47	3494198	0.00	0.76
PyGithub/PyGithub	0	0	0	298500	-	-	0	29494	0	269006	0.00	-
pyparsing/pyparsing	0	0	0	561000	-	-	0	39489	0	521511	0.00	-
psf/requests	0	0	1	176999	-	0.00	1	11831	0	165168	0.00	1.00
saltstack/salt	0	0	0	3759000	-	-	0	1787	0	3757213	0.00	-
encode/starlette	0	0	0	267900	-	-	0	6400	0	261500	0.00	-
vertexproject/synapse	0	0	26	342574	-	0.00	0	62	26	342512	0.00	0.00
twisted/twisted	1	2	30	2866767	0.33	0.03	3	14111	28	2852658	0.00	0.10
urllib3/urllib3	0	3	17	515680	0.00	0.00	16	12545	1	503138	0.00	0.94
xonsh/xonsh	0	1	30	1533569	0.00	0.00	0	12024	30	1521546	0.00	0.00
Aggregate	68	346	641	30936145	0.16	0.10	569	381483	140	30555008	0.00	0.80

4 Discussion

4.1 Causes of Test Flimsiness

Following our empirical analysis, we manually inspected a random sample of mutant-test pairs that showed statistically significant evidence of test flimsiness. In each case, we looked at the mutant diff, the source code of the test case, the traceback from a random flaky failure, and the code under test in the project's GitHub repository. Each of the four authors of this paper reviewed up to 30 examples independently before meeting as a group to discuss findings. As external researchers rather than project developers, we cannot conclusively establish the root causes of specific instances of flimsiness using the methods presented in this paper. A more systematic root cause analysis, potentially involving project developers, is left for future work. Nevertheless, our exploratory investigation revealed some likely causes, which we discuss in this section (one example is already given in Figure 1). All analyzed examples are included in the replication package [5] for further inspection.

Figure 2 shows an example from the cylc-flow project. The test case checks that a command-line tool lists items (called flows) in alphabetical order by name when sorting is enabled. The code under test implements natural sorting, which means that numbers within names are compared numerically (for example, `item2` comes before `item10`). The mutation changed an equality check (`==`) to a non-identity check (`is not`), causing the sort-key function to mistakenly remove the final part of some names. This made different names sometimes produce identical sort keys, so their relative order after sorting depended on the order in which directory entries were read from the file system, which is inherently non-deterministic.

Figure 3 shows an example from the cirq project. This test case checks whether a benchmarking routine produces the expected error rates when simulating noisy two-qubit quantum circuits. The mutation changed the numerical range used to generate single-qubit rotation gates, altering the distribution of randomly constructed circuits on which the benchmark operates. Even with a fixed random

```
     ret = []
     for item in _NAT_SORT_SPLIT.split(key):
         for fcn in fcns:
             with suppress(TypeError, ValueError):
                 ret.append(fcn(item))
                 break
-    if ret[-1] == '':
+    if ret[-1] is not '':
         ret.pop(-1)
     return ret
```

(a) An excerpt from the mutant diff.

```
async def test_name_sort(flows, mod_test_dir):
    """It should sort flows by name."""
    # one stopped flow
    opts = ScanOptions(states='all', sort=True)
    lines = []
    await main(opts, write=lines.append, scan_dir=mod_test_dir)
    assert len(lines) == 4
    assert '-paused-' in lines[0]
    assert '-running-' in lines[1]
    assert '-stopped-' in lines[2]
    assert 'a/b/c' in lines[3]
```

(b) The test case source code.

Figure 2: An example of flimsiness from cylc-flow.

```
     rs = value.parse_random_state(random_state)
-    exponents = np.linspace(0, 7 / 4, 8)
+    exponents = np.linspace(0, 7 / 5, 8)
     single_qubit_gates = [
         ops.PhasedXZGate(...)
         for a, z in itertools.product(exponents, repeat=2)
     ]
     return [
         random_rotations_between_two_qubit_circuit(
             ...
             single_qubit_gates=single_qubit_gates,
             seed=rs,
         )
         for _ in range(n_library_circuits)
     ]
```

(a) An excerpt from the mutant diff.

```
def test_parallel_two_qubit_xeb(...):
    res = cirq.experiments.parallel_two_qubit_xeb(
        sampler=sampler,
        qubits=qubits,
        n_repetitions=100,
        n_combinations=1,
        n_circuits=1,
        cycle_depths=[3, 4, 5],
        random_state=0,
    )
    got = [
        res.xeb_error(*reversed(pair))
        for pair in res.all_qubit_pairs
    ]
    np.testing.assert_allclose(got, 0.1, atol=1e-1)
```

(b) An excerpt from the test case source code.

Figure 3: An example of flimsiness from cirq.

seed, these changes affect the statistical properties of the simulated results, leading to variations in the estimated error rates. Because the test case asserts that these rates must be close to 0.1 within a tight tolerance, some runs now exceed that threshold.

4.2 Predicting Flimsiness from Code Context

We examined whether experts and large language models (LLMs) with software engineering knowledge can predict if a mutation will induce flimsiness based solely on the change and its surrounding code context. Since practicing software engineers commonly use LLMs in their work [28], this task was conceived of as an exploratory probe to contextualize the findings of Section 3, rather than a human-to-LLM comparison. From each subject, we randomly sampled one mutant that induced test flakiness in at least one test case and one that did not, excluding any previously inspected mutants, yielding 18 in total. With no prior knowledge of which mutants induced flakiness, the four authors of this paper independently reviewed each mutant's diff and responded with "true," "false," or "unsure" to the prompt: "I think this mutation could induce flakiness in a previously stable test case in this project." Each author also provided a short justification. We further instructed OpenAI's GPT-4.1 to perform the same independent classification as an additional rater to support the qualitative analysis, rather than as a comparison benchmark for human performance. To increase determinism, we set the model's temperature to 0 and further saved all the responses in the replication package [5] for transparency.

Table 5 presents the results from this exploration. We treated rater responses as predicted labels and compared them with the true labels from RQ1, aggregating predictions by majority vote while excluding "unsure" responses; ties were assigned "unsure." This yielded definitive outcomes for 17 of the 18 mutants, of which 12 were correctly classified (71%). These results suggest that predicting flimsiness from code context alone is difficult and that its causes are often subtle and dependent on broader factors not visible in the diff of the mutated program's source code. This reinforces our interpretation of the RQ4 results, namely that the causes of flimsiness are nuanced. Closer inspection of individual cases in Figure 4 illustrates this variety. For Sampled Mutant #5, all raters correctly predicted induced flakiness, citing a concurrency or multiprocessing context, which is commonly associated with flakiness [18, 33, 37]. For Sampled Mutant #9, all raters correctly judged the mutation as non-flimsy, noting that it would trigger a `TypeError` in a `__repr__` method, which is unlikely to execute non-deterministically. In contrast, for Sampled Mutant #12, all human raters overlooked that the change could induce flakiness by sharing mutable state between tests, whereas GPT-4.1 recognized this possibility.

4.3 Implications and Future Directions

This paper's study is the first to characterize test flimsiness. Its findings have major implications for developers and researchers, and therefore open a multitude of avenues for future work.

Impact on Mutant-Driven Techniques. In 54% of the studied subjects, our statistical analysis reliably identified instances where code-under-test mutations induced flakiness in previously stable test cases. This phenomenon poses a previously unrecognized challenge to techniques that rely on mutation operators, such as fault localization [36, 55, 56], regression testing [19], performance testing [45], and web testing [32]. Prior work has considered the influence of pre-existing flaky tests [8, 48], but has not addressed the risk of mutation-induced flakiness introduced during analysis. To our knowledge, this is the first study to systematically characterize

Table 5: The four authors and GPT-4.1 classified 18 mutants using the prompt: "I think this mutation could induce flakiness in a previously stable test case." Responses (T: True, F: False, U: Unsure) were treated as predicted labels and compared with the true labels from RQ1 (True). The table shows individual responses, majority vote, and the true labels.

	Sampled Mutant ID																	
Labels	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
Auth. 1	F	T	F	F	U	T	T	T	F	F	F	T	F	T	U	F	F	F
Auth. 2	F	T	F	T	F	T	F	F	T	F	F	F	F	T	T	F	F	T
Auth. 3	F	T	T	F	F	T	F	F	F	F	F	T	F	T	F	F	T	T
Auth. 4	T	F	F	F	F	T	F	F	F	F	T	T	F	T	F	F	F	F
GPT-4.1	F	F	T	T	T	T	F	T	T	F	T	F	T	T	T	F	F	T
Vote	F	T	F	F	F	T	F	F	F	F	F	T	F	T	U	F	F	T
True	F	T	F	T	F	T	F	T	T	F	F	T	T	T	T	F	F	F

```
        with self._core.lock:
            handlers = self._core.handlers.copy()
-           for handler in handlers.values():
+           for handler in []:
                handler.complete_queue()
                tasks.extend(handler.tasks_to_complete())
```

(a) Sampled Mutant #5 (All voted True; true label was True.)

```
    def __repr__(self):
-       return "(id=%r, name=%r)" % (self.id, self.name)
+       return "(id=%r, name=%r)" / (self.id, self.name)
```

(b) Sampled Mutant #9 (All voted False; true label was False.)

```
    def __init__(self, shell, outputs=None):
        self.shell = shell
-       if outputs is None:
+       if not outputs is None:
            outputs = []
        self.outputs = outputs
```

(c) Sampled Mutant #12 (Authors voted False; true label was True.)

Figure 4: Code diff excerpts from three sampled mutants.

such flakiness. Future work should assess its practical consequences within each mutant-driven technique's specific evaluation framework, where it may distort established measures of effectiveness.

Risk of Overlooked Bugs. For developers, flakiness is known to complicate bug detection by obscuring fault signals [39]. A dangerous consequence is that developers may begin to ignore flaky test failures and subsequently miss real bugs [44]. Since test flimsiness is a form of flakiness triggered by mutations, which are expected to be coupled with real bugs [29, 43], it could potentially arise precisely when a bug is introduced. Given that developers may disregard flaky test failures [25], this could increase the likelihood of bugs being overlooked. While our study does not evaluate this risk directly, it exposes an important and urgent need for future work to determine whether test flimsiness heightens the likelihood of real bugs being overlooked in practice.

Implications for Flakiness Detection. We found that the flaky tests detected by the mutation-based rerunning strategy are especially unlikely to be detected by the standard strategy. As shown in Table 3, its median yield is also far higher: 740 flaky tests versus 163. Prior work has perturbed the execution environment to augment rerunning [50, 51], or the test code to support benchmarking in the face of limited real-world datasets [13, 15]. Mutation-based rerunning complements these efforts by perturbing a third major source of flakiness: the code under test [18, 22, 33]. Crucially, this strategy uses standard, off-the-shelf mutation tools, requiring no custom operators or special instrumentation, making it easy to adopt in existing workflows. While we do not claim the resulting flakiness fully mirrors that seen in the wild, our results show that it effectively amplifies detection. In particular, it could be used to expand flaky test datasets with failures that standard rerunning misses. Future studies could explore combining mutation-based rerunning with environment and test code perturbations to better stress-test flaky test mitigation techniques.

Developer Attitudes. This study was primarily based on statistical analysis of test execution data. Industrial software developers were not directly involved in the methodology at any stage. Understanding how developers perceive test flimsiness, and to what extent they recognize its adverse effects, is crucial for evaluating its practical impact and informing effective mitigation strategies. Given that mutation testing is already used in practice [46], it is particularly important to understand whether developers encounter, or could detect, flimsiness during routine use of mutation tools. Future studies should conduct developer surveys and interviews to assess awareness of test flimsiness, reactions to its implications, and the perceived usefulness and feasibility of mutation-based rerunning.

Mutation Operator Characteristics. Our analysis revealed substantial variation across mutation operators in their tendency to produce mutants that induce test flakiness. Operators that more frequently generate mutants killed by test cases also tend to be more effective at inducing flakiness. Although this correlation may be expected, it remains noteworthy because it highlights flimsiness as an inherent side effect of standard mutation operators. This underscores both the danger, as a potential threat to the validity of mutant-driven techniques, and the utility, as a means of surfacing hidden flakiness. Future work could investigate whether certain operators consistently uncover more flaky tests across a broader range of projects, and whether other mutant characteristics, such as the surrounding code context where mutations are applied, can be leveraged to prioritize mutants more likely to expose flakiness.

Understanding the Causes of Test Flimsiness. Our results indicate that non-deterministic coverage of mutated lines prior to mutation does not reliably predict whether a mutant will induce test flakiness. This finding suggests that the root causes of test flimsiness are more complex and nuanced than simply manifesting unstable coverage. A more thorough empirical investigation into these causes could guide the development of focused strategies to reduce flimsiness. Future work could investigate factors beyond coverage, such as the presence of existing code smells, properties of the affected test cases, and the impact on timing and concurrency-related behaviors in the mutated code under test.

5 Related Work

Shi et al. [48] studied how non-deterministic coverage (variation in the set of statements executed across repeated runs of passing test cases) can undermine the reliability of mutation testing results. They showed that such instability can cause mutants to be misclassified as killed or survived even when test outcomes remain constant, and proposed rerunning tests and executing them in isolation to reduce this noise. In contrast, we examined how mutations themselves can induce non-deterministic outcomes in otherwise stable test cases, revealing a different causal direction between mutation testing and flakiness. Where their work treats coverage instability as the primary indicator of flakiness, our RQ4 results show that pre-mutation coverage instability is a poor predictor of flimsiness.

Alshammari et al. [8] conducted an empirical study examining the impact of pre-existing flaky tests on mutation testing. They highlighted that flaky tests introduce significant uncertainty in the cause behind killed mutants, raising doubt about whether a mutant was killed reliably or only incidentally due to flakiness. Their analysis of 22 Java projects revealed that 19% of mutants killed by pre-existing flaky tests were, in fact, unreliably killed. To address this concern, they investigated a lightweight failure deduplication approach, which they confirmed to be effective. While Alshammari et al. focused on the effects of pre-existing flakiness on mutation testing, our work addresses a distinct and critical gap: the induction of new flakiness by mutation operators themselves.

Chen et al. [13] introduced Croissant, a framework designed to inject artificial flakiness into JUnit test suites for the purposes of evaluating flakiness detection and mitigation techniques. They devised a defect model of test flakiness, comprising of 18 flaky anti-patterns, based on their analysis of 330 GitHub issues related to flakiness and a range of papers on the topic. They implemented 18 special mutation operators designed to inject such anti-patterns directly into test code, simulating order-dependent, non-order-dependent, and implementation-dependent flakiness. Their empirical evaluation demonstrates its effectiveness, particularly in its ability to expose bugs in iDFlakies [30], a state of the art flaky test detection tool. Unlike Croissant, which focuses on injecting artificial flakiness into test code via custom mutation operators, our work investigates the previously overlooked phenomenon of test flimsiness. The key distinguishing factor is that flimsiness describes flakiness that arises from mutations to the code under test, rather than the test code, produced by standard mutation operators as found in out-of-the-box mutation testing tools [1], not purpose-built frameworks. As such, flimsiness highlights a threat to the reliability of mutant-driven techniques, including mutation testing.

Cordy et al. [15] presented FlakiMe, a Maven plugin for injecting controlled flakiness into Java test suites to assess the impact on testing techniques, including mutation testing and automated program repair. It operates by instrumenting test bytecode to probabilistically raise unchecked exceptions at specified "flake points." The probability of failure is determined by a user-defined flakiness prediction model. Cordy et al. used FlakiMe to assess the impact of flakiness on mutation testing, finding that it inflates mutation scores, and automated program repair, finding that it reduces the number of valid patches. As with Croissant, FlakiMe injects artificial flakiness into the test cases directly, which distinguishes it from our study that focuses on off-the-shelf mutation operators applied to the source code of the program under test. Furthermore, the flakiness produced by FlakiMe is a controlled input for experimentation, whereas our notion of test flimsiness is a discovered phenomenon that arises from the mutation process itself.

Habchi et al. [24] proposed FLAKER, a technique to generate order-dependent flaky tests via mutation. It deletes "helper statements" that usually stabilize shared state between test cases [49]. The primary motivation is to inject a specific type of artificial flakiness (order-dependency) into test code to generate flaky test datasets. In that sense, the approach is similar Croissant, along with all the characteristics that distinguish it from our work.

Prior work has introduced the term *fragile* to describe test cases that fail unexpectedly when small changes are made to the application under test, even though the core functionality being tested remains correct [14]. This is particularly problematic in web testing, where even minor alterations in web pages can easily break existing test code, rendering it unable to correctly locate and interact with web page elements [16]. This contrasts with our work, as test fragility relates to deterministic, implementation-coupled test failures. Test flimsiness, however, is characterized by its non-determinism: it defines the phenomenon where a previously stable test case begins to fail intermittently (i.e., becomes flaky) as a direct result of a mutation to the source code under test.

6 Conclusion and Future Work

This paper characterizes flimsiness, a prevalent phenomenon where stable test cases become flaky due to standard mutation operators applied to the code under test. This distinct manifestation of flakiness, observed in over half the investigated open-source projects, poses a fundamental threat to the reliability of mutation testing. It also endangers other mutant-driven techniques that implicitly assume deterministic test outcomes in response to code-under-test mutations. In addition to posing a challenge to widely adopted software engineering tools and techniques, flimsiness offers a promising new lens for detecting potential flakiness. By demonstrating how perturbing the code under test enhances rerunning-based flaky test detection, our study extends prior work that perturbs the execution environment or test code, thereby building upon a suite of techniques that target the major sources of flakiness. We plan to investigate all future work outlined in Section 4.3. For instance, along with studying flimsiness in subjects implemented in different programming languages, in future work we will empirically assess the practical consequences of flimsiness across the wide range of mutant-driven techniques that aid software engineers.

Acknowledgments

Owain Parry and Phil McMinn are supported by the EPSRC grant "Test FLARE" (EP/X024539/1).

References

[1] 2025. Cosmic Ray: mutation testing for Python — Cosmic Ray documentation. https://cosmic-ray.readthedocs.io/en/latest/.
[2] 2025. Coverage.py — Coverage.py 7.9.2 documentation. https://coverage.readthedocs.io/en/7.9.2/.
[3] 2025. Docker: Accelerated Container Application Development. https://www.docker.com/.
[4] 2025. ossf/criticality_score: Gives criticality score for an open source project. https://github.com/ossf/criticality_score.

[5] 2025. Replication Package. https://doi.org/10.15131/shef.data.30428569.v1.

[6] 2025. SciPy. https://scipy.org/.

[7] A. Alshammari, P. Ammann, M. Hilton, and J. Bell. 2024. 230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 257–268.

[8] A. Alshammari, P. Ammann, M. Hilton, and J. Bell. 2024. A Study of Flaky Failure De-Duplication to Identify Unreliably Killed Mutants. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 257–262.

[9] A. Alshammari, C. Morris, M. Hilton, and J. Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1572–1584.

[10] Y. Benjamini and D. Yekutieli. 2001. The Control of the False Discovery Rate in Multiple Testing Under Dependency. *Annals of Statistics* (2001), 1165–1188.

[11] C. Bonferroni. 1936. Teoria Statistica Delle Classi E Calcolo Delle Probabilita. *Pubblicazioni Del R Istituto Superiore Di Scienze Economiche E Commericiali Di Firenze* 8 (1936), 3–62.

[12] R. D. Boschloo. 1970. Raised Conditional Level of Significance for the 2×2-Table When Testing the Equality of Two Probabilities. *Statistica Neerlandica* 24, 1 (1970), 1–9.

[13] Y. Chen, A. Yildiz, D. Marinov, and R. Jabbarvand. 2023. Transforming Test Suites into Croissants. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 1080–1092.

[14] R. Coppola, M. Morisio, and M. Torchiano. 2018. Mobile GUI Testing Fragility: A Study on Open-Source Android Applications. *Transactions on Reliability* 68, 1 (2018), 67–90.

[15] M. Cordy, R. Rwemalika, A. Franci, M. Papadakis, and M. Harman. 2022. FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 982–994.

[16] S. Di Meglio and L. L. L. Starace. 2024. Towards Predicting Fragility in End-to-End Web Tests. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 387–392.

[17] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 254–264.

[18] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 830–840.

[19] S. Elbaum, A. G. Malishevsky, and G. Rothermel. 2000. Prioritizing Test Cases for Regression Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 102–112.

[20] M. Gruber and G. Fraser. 2022. A Survey on How Test Flakiness Affects Developers and What Support They Need to Address It. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 82–92.

[21] M. Gruber and G. Fraser. 2023. FlaPy: Mining Flaky Python Tests at Scale. In *Proceedings of the International Conference on Software Engineering Companion (ICSE-C)*. 127–131.

[22] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 148–158.

[23] M. Gruber, M. F. Roslan, O. Parry, F. Scharnböck, P. McMinn, and G. Fraser. 2024. Do Automatic Test Generation Tools Generate Flaky Tests?. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1–12.

[24] S. Habchi, M. Cordy, M. Papadakis, and Y. Le Traon. 2021. On the Use of Mutation in Injecting Test Order-Dependency. *arXiv preprint arXiv:2104.07441* (2021).

[25] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon. 2022. A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 244–255.

[26] M. Harman and P. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–23.

[27] M. Hilton, J. Bell, and D. Marinov. 2018. A Large-Scale Study of Test Coverage Evolution. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 53–63.

[28] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8, Article 220 (Dec. 2024).

[29] Y. Jia and M. Harman. 2010. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2010), 649–678.

[30] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. IDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 312–322.

[31] R. Lawson. 2004. Small Sample Confidence Intervals for the Odds Ratio. *Communications in Statistics-Simulation and Computation* 33, 4 (2004), 1095–1113.

[32] M. Leotta, D. Paparella, and F. Ricca. 2024. Mutta: A Novel Tool for E2E Web Mutation Testing. *Software Quality Journal* 32, 1 (2024), 5–26.

[33] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. 643–653.

[34] S. Lydersen, M. W. Fagerland, and P. Laake. 2009. Recommended Tests for Association in 2× 2 Tables. *Statistics in Medicine* 28, 7 (2009), 1159–1175.

[35] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60.

[36] M. Papadakis and Y. Le Traon. 2014. Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach. In *Proceedings of the Symposium on Applied Computing (SAC)*. 1293–1300.

[37] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2021. A Survey of Flaky Tests. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–74.

[38] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 93–104.

[39] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. Surveying the Developer Experience of Flaky Tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 253–262.

[40] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. 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)*. 160–164.

[41] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2023. Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models. *Empirical Software Engineering* 28, 3 (2023).

[42] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2025. Systemic Flakiness: An Empirical Analysis of Co-Occurring Flaky Test Failures. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*.

[43] G. Petrović, M. Ivanković, G. Fraser, and R. Just. 2021. Does Mutation Testing Improve Testing Practices?. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 910–921.

[44] M. T. Rahman and P. C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated With Firefox Builds. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 857–862.

[45] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura. 2018. Search-Based Mutation Testing To Improve Performance Tests. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO-C)*. 316–317.

[46] A. B. Sánchez, J. A. Parejo, S. Segura, A. Durán, and M. Papadakis. 2024. Mutation Testing in Practice: Insights From Open-Source Software Developers. *IEEE Transactions on Software Engineering* 50, 5 (2024), 1130–1143.

[47] P. Schober, C. Boer, and L. A. Schwarte. 2018. Correlation Coefficients: Appropriate Use and Interpretation. *Anesthesia & Analgesia* 126, 5 (2018), 1763–1768.

[48] A. Shi, J. Bell, and D. Marinov. 2019. Mitigating the Effects of Flaky Tests on Mutation Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 112–122.

[49] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-dependent Flaky Tests. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 545–555.

[50] D. Silva, M. Gruber, S. Gokhale, E. Arteca, A. Turcotte, M. d'Amorim, W. Lam, S. Winter, and J. Bell. 2024. The Effects of Computational Resources on Flaky Tests. *IEEE Transactions on Software Engineering* 50, 12 (2024), 3104–3121.

[51] D. Silva, L. Teixeira, and M. D'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. 301–311.

[52] C. Spearman. 1904. The Proof and Measurement of Association Between Two Things. *The American Journal of Psychology* 15, 1 (1904), 72–101.

[53] B. Vancsics, T. Gergely, and A. Beszédes. 2020. Simulating the Effect of Test Flakiness on Fault Localization Effectiveness. In *Proceedings of the International Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. 28–35.

[54] A. J. Vickers. 2005. Parametric Versus Non-Parametric Statistics in the Analysis of Randomized Trials With Non-Normally Distributed Data. *BMC Medical Research Methodology* 5, 1 (2005), 35.

[55] B. Wang, J. Wei, M. Chen, C. Chen, Y. Lin, and J. M. Zhang. 2025. A Systematic Exploration of Mutation-Based Fault Localization Formulae. *Software Testing, Verification and Reliability* 35, 1 (2025), e1905.

[56] Y. Yan, S. Jiang, Y. Zhang, and C. Zhang. 2023. An Effective Fault Localization Approach Based on PageRank and Mutation Analysis. *Journal of Systems and Software* 204 (2023), 111799.