Mitigating the Effects of Flaky Tests on Mutation Testing

August Shi
University of Illinois
Urbana, IL, USA
awshi2@illinois.edu

Jonathan Bell
George Mason University
Fairfax, VA, USA
bellj@gmu.edu

Darko Marinov
University of Illinois
Urbana, IL, USA
marinov@illinois.edu

ABSTRACT

Mutation testing is widely used in research as a metric for evaluating the quality of test suites. Mutation testing runs the test suite on generated *mutants* (variants of the code under test) where a test suite *kills* a mutant if any of the tests fail when run on the mutant. Mutation testing implicitly assumes that tests exhibit deterministic behavior, in terms of their coverage and the outcome of a test (not) killing a certain mutant. Such an assumption does not hold in the presence of *flaky tests*, whose outcomes can non-deterministically differ even when run on the same code under test. Without reliable test outcomes, mutation testing can result in unreliable results, e.g., in our experiments, mutation scores vary by four percentage points on average between repeated executions, and 9% of mutant-test pairs have an unknown status. Many modern software projects suffer from flaky tests. We propose techniques that manage flakiness throughout the mutation testing process, largely based on strategically re-running tests. We implement our techniques by modifying the open-source mutation testing tool, PIT. Our evaluation on 30 projects shows that our techniques reduce the number of "unknown" (flaky) mutants by 79.4%.

CCS CONCEPTS

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

KEYWORDS

Flaky tests, mutation testing, non-deterministic coverage

ACM Reference Format:
August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the Effects of Flaky Tests on Mutation Testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330568

1 INTRODUCTION

Software testing is an important part of the software development process that helps improve software quality. However, software testing is only effective if the test suites are of high quality in terms of their bug-finding capability. Having a large number of tests in the test suite or having high code (statement) coverage does not necessarily indicate the test suite has a high bug-finding capability.

Mutation testing is widely used in research as a better measure of the quality of test suites [19]. The idea of mutation testing is to first use mutation operators, which introduce small syntactic changes into the code under test, to create variants of the original code called *mutants*. Next, mutation testing runs the test suite on each mutant. If the test suite, originally with all tests passing on the original code under test, has any test that fails now on the mutant, then the mutant is considered *killed*. The output of mutation testing is a *mutation score*, the percentage of all generated mutants that are killed by the test suite. Mutation scores are most widely used to compare testing techniques [19], but researchers have also used mutation testing in other areas, e.g., to guide and evaluate test generation [10], test-suite reduction [38, 40], or test prioritization [24, 26]. Furthermore, practitioners are starting to apply mutation testing on real-world code [1, 33, 34]. Small differences in mutation scores can affect the overall conclusion, e.g., whether one technique is better than another.

The results of mutation testing become *unreliable* in the presence of *flaky tests*, which can exhibit different behaviors (e.g., passing or failing) even with no changes to the code under test. Flaky tests have been widely recognized as a significant challenge in regression testing, where (ideally) only a regression should cause a test to fail [6, 12, 16, 25, 45]. Traditional mutation testing implicitly assumes that test behavior is deterministic while rerunning the same tests on different mutants. If a test can have different outcomes when run on the same mutant, then it is unclear if the test actually kills the mutant. In particular, if a test has non-deterministic coverage [17, 29], a mutant can appear "killed" by a test in the traditional sense (the test fails when run on the mutant), yet that test *does not even execute* the mutated part of the code; we say then that the test does not cover the mutant. Non-deterministic coverage is not necessarily bad; such non-determinism may even be intended or hard to control, particularly if the code under test has concurrency or other non-determinism. However, a test that fails yet does not cover a mutant should not be considered to have killed the mutant.

We perform an empirical study to assess the impact of flaky tests on mutation testing. We measure how often the set of lines covered by each test changes between multiple test runs, which can provide an indication (although perhaps an underestimation) of the flakiness of each test. This coverage map is used to guide the generation of mutants, mutating only the statements covered by each test. When running each mutant-test pair, we also track whether the mutant is covered: due to non-determinism, some mutants may not in fact be covered when tested, even though the prior coverage collection indicated that it should be.

Rather than follow traditional mutation testing and declare these uncovered mutants as "killed" (if the test fails) or "survived" (if

ISSTA '19, July 15–19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330568

the test passes), we now refer to the status of such mutants as *"unknown"*. The status of these mutant executions are unknown because the mutant is not covered during the execution; if there is no test execution that covers the mutant, then the overall status of that mutant is also unknown. Although this approach is not guaranteed to find all unknown mutants, it has no false positives—all mutants reported as "unknown" should be truly treated as unknown. From our experiments, we find that about 4% of mutants are unknown, while 9% of test executions on mutants have an unknown status. The differences might appear small, but such differences can misguide techniques that rely on mutant detection.

We propose a set of changes to traditional mutation testing to increase the reliability of mutation testing in the presence of flaky tests. First, we propose to obtain stable coverage for each test, guiding mutation generation to generate more mutants that can be covered by the tests. Second, we propose, when running tests on each mutant, to rerun tests that do *not* cover the mutant when they should be able to cover the mutated part based on earlier coverage collection; by rerunning until the test covers the mutant we can obtain a more reliable result for the execution of the test on the mutant. We also propose using different isolation strategies to increase the likelihood of the test covering the mutant. Finally, we propose prioritizing tests run (and rerun) per mutant as to ensure both faster and more reliable mutation testing results.

This paper makes the following contributions:

- **Motivating Study:** We show that even for tests that do not *appear* flaky in terms of the test outcome, the coverage of each test can often vary non-deterministically.
- **Resilient Mutation Testing:** We propose techniques to improve the resilience of mutation testing that: (1) improve coverage collection to inform mutant generation of more possible mutants, (2) force reruns of tests on mutants not covered to obtain reliable mutant killing results, and (3) prioritize tests for faster yet also more reliable mutation testing.
- **Evaluation:** We evaluate the improvements from these techniques, yielding concrete suggestions for mutation testing tool builders (and users) for improving mutation testing.
- **Tool and dataset:** We make all our modifications to PIT, our dataset of coverage, test results, and mutation results publicly available for other researchers [37].

2 BACKGROUND AND MOTIVATION

Most prior research on flaky tests has focused on detecting when the outcome (passing or failing) of a test is flaky [6, 39, 45]. However, in the context of mutation testing, a flaky test might impact the mutation testing process even if its flakiness does *not* change the result of a test. More subtle variations in program behavior — such as the set of statements covered by a test, or the path that the test follows — might vary from execution-to-execution. These tests might never appear flaky to a developer (if they always pass), but nonetheless can result in non-deterministic mutation testing results.

```
 1 public void run() {
 2   long startTime = System.currentTimeMillis();
 3   boolean isWaiting;
 4   synchronized (this) {
 5     long timeLeft = timeout -
 6       (System.currentTimeMillis() - startTime);
 7     isWaiting = timeLeft > 0;
 8     while (!stopped && isWaiting) {
 9       try {
10         wait(timeLeft);
11       } catch (final InterruptedException e) {
12       }
13       timeLeft = timeout -
14         (System.currentTimeMillis() - startTime);
15       isWaiting = timeLeft > 0;
16     }
17   }
18   // notify the listeners outside of the
19   // synchronized block (see EXEC-60)
20   if (!isWaiting) {
21     fireTimeoutOccured();
22   }
23 }
```

Listing 1: Example demonstrating flakiness in coverage

Consider the code in Listing 1, which exhibits non-determinism in the Apache Commons-Exec project (taken from the class `Watchdog` from revision `39250d73`). Many tests call this method, running it in a separate thread to observe whether a task has finished within a timeout window — if not, and the task has not yet timed out, then the `Watchdog` waits and tries again. If the task finishes *before* the run method is called, then the first evaluation of line 8 evaluates to `false`, and the loop body is not executed (lines 9-15). Note that the code is correct regardless of whether lines 9-15 are executed; the execution of the loop body only indicates that the `Watchdog` will wait and check again later. To understand the effect of this specific non-determinism, we execute every test in this project ten times, recording how often each statement is covered by each test. In this case, there are nine tests that cover these lines, but of those nine, only four tests cover these lines *all* ten times — the rest demonstrate flaky behavior, not executing these lines in all executions.

Since these lines might be non-deterministically (flakily) covered by each test, a *mutation* that is applied to any of these lines might never even be executed. It is easy to imagine a test that would kill a mutant that mutated lines 9-15 *if and only if* the test executes those lines. Hence, on some executions, the mutant is killed (because it is covered) and on others it survives (because it is not covered).

2.1 Motivating Study

While recent work has examined how often statement coverage changes between *different* versions of code [17], no prior work has considered the flakiness of coverage in a *single* version of a program. To gauge the scope of this problem, we analyze the statement coverage of 30 open-source projects selected from projects recently studied in the context of flaky tests [6, 45].

For each project, we collect the statement coverage of each test using the popular Java mutation testing tool, *PIT* [7, 8]. Then, we collect the coverage of each test *16* more times, isolating each test run in its own JVM, to reduce flakiness imposed by test-order dependencies [4, 5, 11, 15, 49]. Prior work [31, 45] found 10 reruns effective at finding flaky tests, and we choose a number at least that many to give us a higher chance of exposing flakiness. We measure non-determinism in coverage as the number of statements with at least one test that does not always cover (or not) that statement.

Table 1 shows the results of this analysis. We observe no test failures in any of the runs, and hence, to the casual observer, there are no flaky tests. However, we find many statements are non-deterministically covered — in the case of *cloudera.oryx*, 74% of

Table 1: Non-determinism in coverage

	Non-determinism at the level of:		
Project	**Statements**	**Tests**	**Pairs**
achilles	1, 634 (13%)	23 (9%)	17, 266 (7%)
assertj-core	757 (4%)	2, 263 (34%)	190, 657 (17%)
cloudera.oryx	12, 251 (74%)	33 (12%)	23, 750 (38%)
commons-collections	133 (1%)	94 (1%)	984 (0%)
commons-configuration	1, 312 (5%)	324 (12%)	42, 564 (1%)
commons-dbcp	1, 266 (14%)	332 (24%)	8, 052 (2%)
commons-exec	541 (40%)	15 (16%)	863 (4%)
commons-functor	555 (7%)	209 (19%)	4, 843 (6%)
commons-io	776 (7%)	104 (8%)	3, 421 (3%)
commons-jxpath	4, 554 (29%)	217 (53%)	173, 425 (20%)
commons-net	388 (6%)	35 (12%)	1, 326 (2%)
commons-validator	1, 870 (25%)	47 (9%)	8, 482 (6%)
cors-filter	0	0	0
dropwizard	2, 911 (19%)	50 (10%)	4, 064 (5%)
empire-db	1, 016 (11%)	20 (17%)	3, 691 (9%)
exp4j	219 (14%)	9 (3%)	1, 122 (1%)
handlebars	20, 872 (40%)	155 (22%)	75, 588 (9%)
handlebars.java	25, 025 (38%)	326 (35%)	191, 759 (14%)
hector	4, 206 (56%)	44 (37%)	21, 284 (41%)
httpcore	7, 017 (40%)	106 (14%)	18, 004 (5%)
jimfs	1, 619 (22%)	43 (10%)	6, 236 (2%)
jsoup	1, 805 (12%)	81 (12%)	36, 488 (4%)
logback	7, 742 (23%)	289 (29%)	43, 169 (7%)
ninja	665 (4%)	48 (13%)	4, 306 (2%)
okhttp	1, 698 (13%)	135 (49%)	5, 691 (1%)
raml-java-parser	1, 499 (8%)	43 (7%)	9, 458 (0%)
retrofit	735 (4%)	91 (14%)	3, 022 (1%)
togglz	464 (2%)	40 (11%)	1, 740 (2%)
undertow	6, 461 (27%)	128 (66%)	58, 360 (18%)
wro4j	3, 365 (10%)	432 (35%)	28, 173 (2%)
30 Total	113, 356 (22%)	5, 736 (16%)	987, 788 (6%)

all statements! We also calculate the number of tests that non-deterministically cover at least one statement, finding that many tests can exhibit non-determinism in coverage, *even though they pass in every run*. This result highlights the applicability of our findings even to projects that appear to have no flaky tests. Finally, we sum the number of statements per-test that are non-deterministically covered. Given each pair of (test, statement), we consider the pair to be non-deterministically covered if that test does not cover that statement in exactly all 17 runs, representing all potential mutation sites that can be non-deterministically executed.

One interesting finding from this study is that the impact of flakiness on statement coverage varies dramatically between projects. For instance, the *cors-filter* project shows deterministic behavior, while the *cloudera.oryx* project shows an extreme amount of non-determinism at the level of statements, and the *hector* project shows the greatest (proportional) amount of non-determinism at the level of statement-test pairs. This variation makes sense, as different projects have different purposes, and some involve more non-determinism: *cloudera.oryx* and *hector* are both client/server programs, and *cors-filter* is a purely deterministic URL filtering utility. Based on this finding, we expect to find a wide range in the impact of flakiness on mutation testing in these (and other) projects.

3 TECHNIQUE

We implement techniques to increase the reliability of mutation testing. While different mutation testing tools have different processes, all are susceptible to flakiness. We implement our techniques by modifying the popular Java mutation testing tool PIT [7].

PIT [7, 8] executes a three-phase process for mutation testing. First, it executes the entire test suite once, collecting the statement coverage of each test. Second, PIT determines what mutants should be generated — given a set of mutation operators and a set of covered basic blocks, PIT can identify candidates for mutation. We say a mutant is covered by a test or set of tests if the test (or at least one in the set) covers the basic block that is to mutated. Typically, a developer's goal is to identify if there exists *any* test that can kill a mutant; if many tests cover that mutant, it may not be necessary to run all of them. In these cases, PIT prioritizes the tests to run, selecting the fastest tests to run first as to decrease the amount of time spent running tests. The final phase is then PIT running the prioritized tests on each mutant, reporting each as killed (a test failed on the mutant), survived (no tests failed on the mutant) or errored (e.g., timeout due to an infinite loop).

3.1 Full Test Suite Coverage Collection

PIT initially runs the test suite and collects which basic blocks each test covers; when PIT later generates mutants, it only generates mutants that have the covered blocks mutated, because tests cannot kill any mutant generated on a block they do not cover. However, as coverage of tests can be flaky, just running the tests once to collect coverage would give an incomplete picture of how many mutants can be killed by the test suite. A mutation testing tool might under-estimate the number of tests that it should execute for a given mutant because in this initial coverage collection phase the test was flaky and did not cover the block targeted for mutation.

We modify PIT's coverage collection phase to run all the tests 16 times and take the union of the blocks covered by the tests run within these 16 times as all the blocks that PIT should later mutate. If a test fails during any of these runs, we remove it from consideration for future phases of the process.

In addition to reruns, we also consider how tests are isolated. When multiple tests share resources, they may be subject to flakiness due to *test-order dependencies* [4, 5, 11, 15, 49]: the behavior of a test might change based on which tests had run previously. To break test-order dependencies, we explore different test isolation strategies: isolating each test in its own JVM, or running all tests in the same JVM. Isolating tests can result in a performance penalty (due to JVM warmup and shared initialization code), and is not (by default) supported by PIT— we add this functionality to PIT.

Note that there are two other alternatives for achieving a similar isolation: using throwaway class loaders (as implemented by the mutation testing tool Major [20]), and using unit test virtualization (as implemented by VmVm [4]). Unfortunately neither approach would work out-of-the-box for the projects in our study (Section 2.1). In the first approach, each test execution is wrapped in its own non-delegating class loader, which ensures that any classes loaded by the test are unloaded at the end (hence breaking in-memory dependencies). This approach is brittle in applications that use their own class loader hierarchies (in particular, common libraries like XStream, as well as OSGI-based component applications). In the second approach, such as through the research tool VmVm, class reloading is emulated through bytecode instrumentation, yielding comparable isolation to executing each test in its own process, but with significantly lower performance overhead. Since our goal is to establish the baseline for the most *reliable* and not necessarily the fastest approach, we did not investigate the application of this research tool further.

Hence, for coverage collection, we consider a total of four different configurations of our modified version of PIT: *1*) *default*: Collect coverage using PIT's normal mechanism (no isolation, no reruns). *2*) *default-reruns*: Collect coverage using PIT's normal mechanism and with reruns (no isolation between tests within a run; multiple reruns where each rerun is isolated from another). *3*) *isolated*: Collect coverage with each test class executed in its own JVM and no reruns. *4*) *isolated-reruns*: Collect coverage with each test class executed in its own JVM and with reruns.

3.2 Mutation Execution

During mutant execution, a flaky test may still not cover a mutant, due to flakiness. We further modify PIT's mutant generation to collect coverage of the mutant. Specifically, at the instruction before where PIT performs the mutation, we insert a call to record whether or not the mutated part of the bytecode is executed during a test run. Using such instrumentation, we collect the set of tests that cover the mutated bytecode. We modify PIT to only consider a test to kill or survive on the mutant if it *also* executes the mutated bytecode at least once.

Tests that do not cover the mutant do not contribute towards the mutant's final status (killed or survived). We introduce a new category, "unknown", for mutant-test pairs (the pair of the mutant and the test that runs on the mutant) where a test that should cover the mutant (based on the initial coverage analysis) but actually does not. For the overall mutant, the same unknown category applies if there are unknown mutant-test pairs corresponding to that mutant, *and* there are no tests that kill and cover the mutant. When categorizing each mutant (overall, based on all mutant-test pairs associated with it), we consider its status to be unknown if at least one mutant-test pair has the "unknown" status, and no mutant-test pairs have the "killed" (and covered) status. For example, if the mutant has at least one survived mutant-test pair, at least one unknown mutant-test pair, but no killed mutant-test pair, that mutant would still have status unknown, because the unknown mutant-test pair(s) could potentially kill the mutant. Note that Major [20], another mutation testing framework, blends these three phases together, and does *not* collect coverage before executing mutants; it would require additional modifications to detect these unknown mutants.

Ideally, there should be no unknown mutant-test pairs, since prior coverage collection shows that each test run for a mutant *can* cover the relevant mutated bytecode. To mitigate the number of unknown mutant-test pairs, we modify PIT to rerun tests that do not cover mutated bytecode on the mutant. For any test that does not cover the mutated bytecode, we schedule to run the test again until it covers the mutated bytecode, up to 16 times. If at any point the test covers the mutated bytecode, we can more reliably record the mutant-test pair's killed or survived status.

Along with re-running tests to reduce the number of unknown mutant-test pairs, we also employ varying degrees of isolation between test executions. By default, PIT compartmentalizes executions into "MutationTestUnits", a collection of mutant-test pairs such that all mutants that have mutations in the same class are grouped into one unit. Each MutationTestUnit execution is in its own JVM. As such, test-order dependencies [49] may be present between tests running against mutations that target the same class.

The risk of test-order dependencies may be much higher with mutants than the original code, because a mutation might inadvertently corrupt some shared state. We call such dependencies *mutant-order dependencies*, because their behavior is typically a result of the mutant, rather than the tests. To combat these mutant-order dependencies, we also consider several more isolation strategies. We modify PIT to also have the option of performing isolation between each execution of each test on each mutant, running each mutant-test pair in its own JVM. We also evaluate isolation at an intermediate level, only performing isolation between test executions on different mutants (all mutant-test pairs for the same mutant are run in the same JVM). We perform up to five reruns at each of these levels in increasing order of overhead.

3.3 Test-Mutant Prioritization

Often, researchers using mutation testing need a full mutation matrix (with the results of running each test on each mutant) for their techniques or evaluation [2, 24, 40]. However, a developer seeking to use mutation testing to evaluate the quality of their test suites may only need a simple mutation score, in which case only the first test that kills a mutant is needed. As such, once a test has killed the mutant, the status of the mutant is known and all subsequent tests for that mutant need not be run. By default, PIT orders tests by their execution time (collected at the same time that coverage is collected). If there are multiple tests that kill a mutant, the fastest-running tests are run first, potentially reducing the total time needed for mutation analysis.

However, a flaky test may not reliably cover the mutant. We enhance PIT's prioritization scheme by *also* considering the observed likelihood of a test to cover each mutant. When repeatedly collecting coverage (Section 3.1), we also record how often each test covers each basic block. Then, when executing tests for each mutant, we prioritize tests that covered the target block the most frequently; these tests are the most likely to cover the mutated bytecode. After prioritizing by the number of times covered, we then prioritize by PIT's default scheme, by execution time.

In addition to prioritizing the order to run tests for each mutant, we also consider two additional strategies during mutation execution concerning how to rerun tests when they do not cover the mutant. The first strategy is to first run all the tests relevant for the mutant, in the order to run them determined before (e.g., by coverage). If none of them kill the mutant but there are tests that did not cover the mutant, rerun those up until a threshold for reruns or until they cover the mutant. This strategy gives an equal chance for all tests to both cover and kill the mutant.

The second strategy is if a test does not cover the mutant, then immediately rerun that test up until the threshold or until it covers the mutant, before running any other tests. The intuition is that if a test does not cover a mutant once, it could be that the test did not cover it by an off-chance: immediately rerunning can give the test another chance to cover. For example, if tests are ordered first by coverage, each test should be more likely to cover the mutant than subsequent tests, so a subsequent run of the same test should still have a better chance of covering the mutant than the remaining tests. Given the different isolation strategies when running tests on mutants (Section 3.2), the immediately rerunning tests strategy only reruns the same test when it does not cover up until the number of reruns remaining before isolating even more.

In both strategies, the basic principle, that it is unnecessary to run any later tests for a mutant if a prior test both covers and kills the mutant, still holds. The goal is to take as little time as necessary to determine for each mutant whether a test reliably kills it.

4 EMPIRICAL STUDY

To measure the impact of flaky tests on mutation testing, we conduct a large-scale evaluation, running our modified version of PIT on 30 open-source Java projects (the same projects presented in Section 2). Our research questions are:

RQ1: How reliable is mutation testing in the presence of flakiness?
RQ2: How should mutation testing tools collect coverage and generate mutants in the presence of flakiness?
RQ3: How should mutation testing tools execute tests on each mutant, in the presence of flakiness?
RQ4: How should mutation testing tools prioritize tests in the presence of flakiness?

4.1 RQ1: Flakiness in Current Mutation Testing

In our background study (Section 2), we find that coverage was often non-deterministic, suggesting that mutation testing may also be susceptible to flakiness. We begin our evaluation by measuring the number of mutants and mutant-test pairs with status "unknown" at the end of a single execution. For each project, we first collect coverage using the approach that we expect to be the most reliable: executing each test in its own process (isolated), and repeating this process 16 times. We collect mutation results for all tests and mutants of all projects using the modified version of PIT and categorized mutants as killed, survived, or unknown (Section 3.2), *without* performing any additional reruns or isolation. We do not consider mutants where PIT reports an error (e.g. timeout or out-of-memory) as PIT's architecture does not allow us to capture any telemetry from mutation executions that entirely crash the JVM.

Table 2 presents the results of our experiment. For each project, we show the number of mutants for each of the three statuses, along with the total number of non-erroring mutants. Overall, 2,866 mutants have status unknown. These mutants have an effect on the potential mutation score. We show the range of the potential mutation scores for each project, given these unknown mutants. The lowest possible mutation score is if all the unknown mutants actually survived, and the highest possible mutation score is if all the unknown mutants are actually killed. As such, the overall mutation score ranges from 77.7% to 82.0%. We carefully investigated these unknown mutants—initially we had found even more—and after patching several bugs in PIT (described in Section 5), confirmed that these mutants are, in fact, unknown.

Table 2 also shows the statuses of mutant-test pairs. Overall, note that mutation scores are quite high across projects, indicating that they likely have high-quality test suites. Nonetheless, we see that 255,194 mutant-test pairs have status unknown. An interesting project to note is how `exp4j` has 0 unknown mutants, yet when examining mutant-test pairs there are actually 302 unknown mutant-test pairs. Overall, our results on mutant-test pairs suggest that a researcher using the full mutation matrix would have a large number of entries in the matrix where the traditional reported result (killed or survived, as reported by PIT) may not be correct, with overall 9% of the mutant-test pairs being unknown.

RQ1: *Mutation testing that does not consider flaky tests can skew results, particularly to researchers aiming to use a full mutation matrix, where overall 9% of mutant-test pairs are unknown. Furthermore, if researchers are relying on mutation scores to improve or evaluate the quality of test suites, then any differences in mutation scores must be beyond the difference we observe, where mutation score can range from 77.7% to 82.0%. If the mutation score does not differ as much as the deviation due to flaky tests, then the difference in scores could very well be due to flakiness noise.*

4.2 RQ2: Coverage Collection Approaches

We evaluate four different coverage collection configurations (Section 3.1), measuring the number and set of mutants PIT generates for each configuration. Table 3 shows overall the number of mutants generated due to PIT finding some test that covers the relevant code for mutation with each coverage collection configuration. The table also shows the number of mutant-test pairs found with each configuration. Interestingly, considering the overall number of mutants, we find that going from default mode with no reruns to the isolation with reruns mode (the most mutants generated) does *not* increase the number of mutants substantially (70,773 to 71,112). Running a Wilcoxon paired rank test for the number of mutants between default mode with no reruns and isolation with reruns results in a p-value of 0.078, suggesting the difference is not statistically significant.The same trend holds for mutant-test pairs, where increasing isolation and reruns leads to more mutant-test pairs, but not by a substantial number. However, a Wilcoxon paired rank test shows that this difference is statistically significant, $p < 0.01$.

We find that in 10 of the projects, some mutants are generated *only* in the non-isolated cases. Examining these mutants in detail, we find that extra mutants generated due to lines that are more likely to be covered during runs or reruns of tests in the same JVM. For example, in `achilles`, the extra mutants generated in the configuration rerunning tests in default mode are due to mutating a `finalize` method, which gets called during garbage collection. Since all other configurations run tests once or in isolation, it is unlikely garbage collection happens during those configurations, and so PIT does not generate mutants for code related to that method.

RQ2: *Collecting mutants generated in the default mode with no reruns configuration may be a good trade-off in getting a significant number of mutants and mutant-test pairs. This observation is good for mutation testing tools, because performing test isolation and rerunning tests can be slow.*

Table 2: Mutations and mutation-test pair statuses, before any flakiness-reduction modifications to PIT.

	Number of Mutants by Status:					Number of Mutant-Pairs by Status:			
Project	Killed	Survived	Unknown	Total	Mut. Score	Killed	Survived	Unknown	Total
achilles	963	112	91	1,166	82.6% – 90.4%	26,612	11,702	268	38,582
assertj-core	4,165	209	13	4,387	94.9% – 95.2%	162,872	119,308	17,983	300,163
cloudera.oryx	532	182	247	961	55.4% – 81.1%	1,414	476	478	2,368
commons-collections	3,199	545	5	3,749	85.3% – 85.5%	30,984	10,064	402	41,450
commons-configuration	4,839	421	19	5,279	91.7% – 92.0%	381,012	277,605	7,286	665,903
commons-dbcp	1,636	927	40	2,603	62.9% – 64.4%	35,125	79,781	494	115,400
commons-exec	207	52	19	278	74.5% – 81.3%	1,945	1,908	397	4,250
commons-functor	1,855	377	11	2,243	82.7% – 83.2%	10,501	6,171	1,233	17,905
commons-io	2,641	384	25	3,050	86.6% – 87.4%	23,930	12,632	1,638	38,200
commons-jxpath	3,308	358	1,092	4,758	69.5% – 92.5%	85,075	38,654	149,988	273,717
commons-net	1,219	405	22	1,646	74.1% – 75.4%	6,747	6,808	82	13,637
commons-validator	1,507	163	10	1,680	89.7% – 90.3%	22,741	14,458	529	37,728
cors-filter	118	23	0	141	83.7% – 83.7%	1,194	1,505	0	2,699
dropwizard	767	212	41	1,020	75.2% – 79.2%	2,436	1,782	255	4,473
empire-db	1,110	1,108	12	2,230	49.8% – 50.3%	4,850	4,506	48	9,404
exp4j	383	8	0	391	98.0% – 98.0%	32,740	8,603	302	41,645
handlebars	1,333	481	65	1,879	70.9% – 74.4%	78,096	55,598	2,825	136,519
handlebars.java	1,585	600	82	2,267	69.9% – 73.5%	132,947	111,926	8,666	253,539
hector	538	354	24	916	58.7% – 61.4%	3,332	1,593	222	5,147
httpcore	2,479	859	67	3,405	72.8% – 74.8%	24,929	26,549	343	51,821
jimfs	1,331	196	47	1,574	84.6% – 87.5%	35,455	14,297	3,996	53,748
jsoup	2,636	622	307	3,565	73.9% – 82.6%	145,634	97,733	46,080	289,447
logback	3,362	767	102	4,231	79.5% – 81.9%	35,058	32,600	8,611	76,269
ninja	895	176	12	1,083	82.6% – 83.7%	6,686	2,177	51	8,914
okhttp	268	41	4	313	85.6% – 86.9%	6,863	2,355	6	9,224
raml-java-parser	2,241	184	0	2,425	92.4% – 92.4%	197,225	86,162	0	283,387
retrofit	979	74	12	1,065	91.9% – 93.1%	23,100	5,321	363	28,784
togglz	694	164	123	981	70.7% – 83.3%	3,334	1,886	447	5,667
undertow	2,653	1,120	330	4,103	64.7% – 72.7%	11,600	20,436	1,284	33,320
wro4j	2,244	841	44	3,129	71.7% – 73.1%	35,221	42,910	917	79,048
30 Total	51,687	11,965	2,866	66,518	77.7% – 82.0%	1,569,658	1,097,506	255,194	2,922,358

Table 3: Flakiness in number of mutants and mutant-test pairs generated with different coverage collection strategies

	Number of Mutants				Number of Mutant-Test Pairs			
Project	Default	Isolated	Default w/ Reruns	Isolated w/ Reruns	Default	Isolated	Default w/ Reruns	Isolated w/ Reruns
30 Total	70,773	70,993	70,877	71,112	3,089,051	3,162,138	3,101,314	3,165,527

4.3 RQ3: Mutant Execution Approaches

Table 4 shows the number of unknown mutants after executing mutant-test pairs with reruns and isolation as described in Section 3.2. The table also shows the reduction in the number of unknown mutants, overall by 2,275, which is 79.4% of the unknown mutants from just running PIT once. The substantial reduction in unknown mutants shows how reruns and isolation help mitigate the effects of flakiness on the final mutant statuses.

When a mutant-test pair is unknown, PIT still records a status for running that test on the mutant even when the test does not cover the mutated bytecode. Given multiple reruns, there are three possibilities: all reruns show the mutant-test pair killed, all reruns show the mutant-test pair survived, and a mix of killed and survived between multiple reruns. We find that, over all mutant-test pairs that are rerun, the three possibilities are roughly equal in occurrence, with 38.0%, 29.7%, and 32.3% of the mutant-test pairs having all reruns being killed, survived, or both, respectively. If we look at the final status of all the mutant-test pairs when they eventually cover the mutation, 51.4% are killed while 48.6% are survived.

To show a deeper breakdown of how reruns and isolation help reduce the number of unknown mutant-test pairs, Table 5 shows for each project the number of additional mutant-test pairs covered after each step of rerunning during mutant execution. Each rerun strategy corresponds to one of the strategies described in Section 3.2, and the columns shows the number of additional covered mutant-test pairs at each rerun iteration. Each additional rerun helps to

Table 4: Flakiness in mutation with reruns and isolation

	Number of Unknown Mutants		
Project	**After Reruns**	**Reduction**	
achilles	21	70	(76.9%)
assertj-core	4	9	(69.2%)
cloudera.oryx	5	242	(98.0%)
commons-collections	0	5	(100.0%)
commons-configuration	12	7	(36.8%)
commons-dbcp	13	27	(67.5%)
commons-exec	18	1	(5.3%)
commons-functor	1	10	(90.9%)
commons-io	20	5	(20.0%)
commons-jxpath	37	1,055	(96.6%)
commons-net	13	9	(40.9%)
commons-validator	7	3	(30.0%)
cors-filter	0	0	(*N/A*)
dropwizard	41	0	(0.0%)
empire-db	6	6	(50.0%)
exp4j	0	0	(*N/A*)
handlebars	17	48	(73.8%)
handlebars.java	11	71	(86.6%)
hector	5	19	(79.2%)
httpcore	19	48	(71.6%)
jimfs	15	32	(68.1%)
jsoup	9	298	(97.1%)
logback	67	35	(34.3%)
ninja	8	4	(33.3%)
okhttp	0	4	(100.0%)
raml-java-parser	0	0	(*N/A*)
retrofit	12	0	(0.0%)
togglz	39	84	(68.3%)
undertow	159	171	(51.8%)
wro4j	32	12	(27.3%)
30 Total	591	2,275	(79.4%)

cover more mutant-test pairs, although the first rerun brings in the most (61,437), with subsequent reruns being less helpful. However, increasing the level of isolation helps more.

Prior work [4] has suggested that the overhead of isolating tests can be quite high (often imposing a slowdown of several times). Our logging of time per rerun strategy, while not quite precise due to the technical difficulties of collecting timing data from experiments executed in the cloud, shows that isolating individual mutant-test pairs per JVM results in overall each mutant-test pair running 52 times slower than for each mutant-test pair at the level of isolating per MutationTestUnit. The large slowdown due to more extreme isolation suggests rerunning tests with the earlier isolation strategies is worthwhile as to reduce as many unknown mutant-test pairs as possible, because the cost of rerunning there is much less than rerunning mutant-test pairs with each one isolated in its own JVM. Overall, we believe that a moderate performance slowdown would still be tolerable to developers, as mutation testing is already a relatively slow activity that is typically not performed as part of the continuous integration process so as to not slow down developer productivity. Moreover, more recent results have shown that this slowdown may be avoidable, while still achieving the same process-level isolation [3].

RQ3: *Switching isolation strategies for running tests on mutants tends to increase the number of mutant-test pairs that are no longer unknown, where later reruns in the same isolation strategy do not bring in as many. This observation suggests the importance of isolation, where some mutants can only be covered with extra isolation. Also, there may not be a need for many reruns per isolation strategy as long as there is a switch in isolation.*

4.4 RQ4: Mutant Prioritization Approaches

Table 6 shows the effectiveness of different test prioritization schemes, with the goal of not just killing mutants faster, but also to *reliably* kill the mutant (by covering it). We use the mutant coverage collected from running tests isolated and with reruns and the mutants killed from reruns (Section 4.3). We simulate the execution time needed to kill all mutants by summing the time for each test executed, using timing obtained from executing tests during PIT's coverage collection phase.

For each project, we show the time to execute tests to reliably kill all mutants (that could be killed in any of our experiments). Since tests have to be rerun when they do not cover the mutant, we sum the time for all executions of the test up until when the test covers the mutant, e.g., if a test covers the mutant on its fifth rerun, then the test needs to be run up to five times before a status for that mutant-test pair is considered, increasing the total testing time by the time to run the test five times. If another test would have killed that mutant on the first execution, then we could have run for a shorter amount of time had we executed that more reliable test first (unless the more reliable test actually takes more time to execute than running the other test those extra five times). For this experiment, we ignore any mutant that is *not* eventually covered and killed by any of its tests (similar to prior work [46]), since these mutants would need to be run the maximum number of times, bloating the total time for executing tests.

We consider five different prioritization schemes to efficiently order tests, including: (1) randomly ordering the tests, (2) prioritizing based on coverage hit count, (3) and the default scheme used by PIT (faster tests prioritized first). For randomly ordering the tests, we randomize the order 10 times and average the performance of all 10 different orders. Following prior work by Zhang et al. [47], we also consider the theoretically (4) best and (5) worst prioritization schemes. The theoretical best scheme reorders test such that a test that kills the mutant is prioritized first. Furthermore, it prioritizes tests that run faster. The theoretical worst scheme prioritizes the slowest tests that do not kill the mutant first. Both schemes are "oracular" in nature, as they can only be computed as a part of post-processing after knowing the mutation testing results and are considered for evaluation purposes only. We also show the time needed by each prioritization scheme for different rerun strategies: either rerunning the test immediately after it first runs, until it covers, or instead to move on to the next test in the order.

From Table 6, we see that prioritizing based on coverage and by PIT's default prioritization scheme result in faster testing times than

Table 5: Efficacy of different rerun strategies in (eventually) covering mutant-test pairs, showing the different isolation strategies, number of reruns performed, and number of additional mutant-test pairs covered.

	Additional mutant-test pairs covered by reruns, isolated by using one JVM per:																
	Mutation Unit					Mutation					Mutant-Test Pair						
Project	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**Total**	**Reduction**
achilles	10	6	7	8	35	20	0	0	0	0	0	0	0	0	0	86	(32.1%)
assertj-core	6,369	77	75	74	74	485	88	91	92	88	9,131	0	0	0	0	16,644	(92.6%)
cloudera.oryx	26	21	24	18	53	211	115	0	0	0	0	0	0	0	0	468	(97.9%)
commons-collections	26	8	2	2	2	1	0	0	0	1	2	0	0	0	0	44	(10.9%)
commons-configuration	791	9	1,662	11	13	38	32	31	34	33	1,174	1	0	1	0	3,830	(52.6%)
commons-dbcp	42	29	13	15	12	12	5	5	2	3	3	0	0	1	0	142	(28.7%)
commons-exec	5	1	1	0	0	0	0	0	0	0	0	0	0	0	0	7	(1.8%)
commons-functor	967	17	14	10	9	12	13	12	13	12	106	0	0	0	0	1,185	(96.1%)
commons-io	41	44	4	4	4	12	9	7	9	5	41	0	0	0	0	180	(11.0%)
commons-jxpath	35,883	27,053	11,679	5,225	17,315	37,257	6,115	290	121	3,461	1,177	0	0	0	0	145,576	(97.1%)
commons-net	28	7	8	3	3	3	0	1	1	0	0	0	0	0	0	54	(65.9%)
commons-validator	7	9	9	9	24	47	39	32	28	27	281	0	0	0	0	512	(96.8%)
cors-filter	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(*N/A*)
dropwizard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(0.0%)
empire-db	16	5	3	0	0	3	0	0	0	0	0	0	0	0	0	27	(56.2%)
exp4j	1	1	1	1	1	14	14	13	12	5	47	0	0	0	0	110	(*N/A*)
handlebars	315	272	191	152	123	581	104	63	29	31	537	0	0	0	0	2,398	(84.9%)
handlebars.java	459	759	414	524	384	2,633	624	330	194	127	1,852	0	0	0	0	8,300	(95.8%)
hector	123	30	2	2	1	0	0	0	0	0	2	0	0	0	0	160	(72.1%)
httpcore	88	18	11	7	5	22	10	9	17	7	73	1	0	0	0	268	(78.1%)
jimfs	1,205	19	276	231	446	159	32	53	23	21	691	0	0	0	0	3,156	(79.0%)
jsoup	14,589	12,700	216	184	143	5,149	6,836	34	18	16	249	0	0	0	0	40,134	(87.1%)
logback	136	67	59	36	50	33	17	13	13	12	142	1	0	2	0	581	(6.7%)
ninja	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	(9.8%)
okhttp	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	6	(100.0%)
raml-java-parser	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(*N/A*)
retrofit	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	4	(1.1%)
togglz	55	10	8	7	9	40	3	3	0	1	5	0	0	0	0	141	(31.5%)
undertow	164	61	40	24	19	56	7	6	6	3	24	2	0	0	0	412	(32.1%)
wro4j	80	77	67	43	36	31	9	7	17	19	57	0	0	0	0	443	(48.3%)
30 Total	61,437	41,302	14,787	6,590	18,762	46,819	14,072	1,000	629	3,872	15,594	5	0	4	0	224,873	(88.1%)

randomly ordering tests. For some projects, the coverage scheme improves somewhat on PIT's default, but offers no benefit in most. Because PIT solely favors running faster tests first, it makes sense that, overall, PIT's default prioritization results in faster testing time. While we see that a good proportion of mutants are non-deterministically covered by *some* test (as suggested in Section 2.1), there are many other tests that do reliably cover the mutant. As such, by waiting to run tests that do not reliably cover the mutant until the end still leads to orders that are similar to what PIT uses by default. Fortunately, all three prioritization schemes perform much closer to the theoretical best than to the theoretical worst.

Concerning rerun strategies, we see that for all five prioritization schemes, it is better to rerun a test immediately if the test does not cover the mutant, which leads to the fewest number of test executions needed to kill a mutant. A Wilcoxon paired rank test between the time for each prioritization scheme per project between the immediate rerun and not immediately rerun versions shows the difference is statistically significant, $p < 0.01$, for all strategies, except for random. Such results suggest that tests that do not cover the mutant once have a high chance of covering it when run again. In other words, while many tests have non-deterministic coverage, they *mostly* cover the same statements.

While we show time for running tests for each prioritization scheme in Table 6, we also compute the number of test executions taken by each prioritization scheme, similar to prior work [47]. Interestingly, we find that randomly ordering tests actually leads to fewer test executions than both ordering by coverage and PIT's default prioritization scheme. These results suggest that research in this area of reordering tests to speed up mutation testing should consider testing time instead of just number of test executions as to better mirror actual mutation testing performance.

> ***RQ4:*** *Prioritizing to rerun a test immediately if it does not cover a mutant gives the test a better chance of covering the mutant than going through all other tests to see if the mutant gets killed.*

Table 6: Estimated time (in seconds) to kill all killable mutants given different prioritization schemes

	Immediately Rerun					Not Immediately Rerun				
Project	**Random**	**Coverage**	**PIT**	**Best**	**Worst**	**Random**	**Coverage**	**PIT**	**Best**	**Worst**
achilles	2,237.2	1,701.9	1,701.9	1,605.9	5,630.7	2,237.2	1,701.9	1,701.9	1,605.9	5,630.7
assertj-core	5,242.6	2,445.6	2,445.6	2,061.0	29,455.2	5,113.9	2,526.3	2,526.3	2,199.8	28,428.3
cloudera.oryx	6,693.9	5,871.9	5,871.9	5,838.3	7,694.9	6,834.1	6,383.1	6,383.1	6,353.8	7,445.6
commons-collections	116.7	39.5	39.5	33.3	418.8	117.0	39.9	39.9	33.7	418.0
commons-configuration	5,507.9	1,153.8	1,154.5	847.2	49,420.7	5,430.1	1,417.9	1,418.5	1,111.4	48,647.6
commons-dbcp	2,571.2	434.4	435.3	153.0	17,647.4	2,571.3	438.0	438.8	156.6	17,644.8
commons-exec	1,178.0	1,124.0	1,211.2	326.5	3,697.3	1,178.0	1,124.0	1,211.2	326.5	3,697.3
commons-functor	105.1	60.6	59.8	56.5	237.3	105.1	60.6	59.8	56.5	237.3
commons-io	499.7	197.8	197.8	167.1	1,391.9	499.7	197.8	197.8	167.1	1,391.9
commons-jxpath	24,979.4	20,890.7	20,890.7	18,726.3	49,481.4	28,305.6	23,863.2	23,863.2	22,398.1	44,766.1
commons-net	1,987.2	1,701.1	1,713.7	1,050.0	3,597.1	1,988.5	1,702.6	1,715.1	1,050.4	3,603.6
commons-validator	301.3	90.2	90.2	57.4	1,513.5	301.5	90.5	90.5	57.7	1,513.5
cors-filter	2.3	0.5	0.5	0.4	12.4	2.3	0.5	0.5	0.4	12.4
dropwizard	457.0	261.3	261.7	230.4	916.1	457.0	261.3	261.7	230.4	916.1
empire-db	589.0	267.7	268.3	219.9	1,267.4	588.3	267.0	267.6	219.9	1,266.4
exp4j	58.3	1.8	1.8	1.1	1,449.2	58.3	1.8	1.8	1.1	1,449.2
handlebars	1,888.6	933.0	932.8	658.9	9,284.0	2,107.2	1,345.3	1,345.1	1,063.9	8,819.4
handlebars.java	11,153.6	6,414.9	6,418.2	4,799.1	46,598.9	13,905.3	10,613.2	10,616.5	9,012.0	40,290.9
hector	293.3	142.5	142.5	137.7	883.1	296.4	146.6	146.6	141.9	883.2
httpcore	1,824.7	392.1	349.6	102.3	5,856.3	1,823.3	392.0	349.6	102.3	5,848.9
jimfs	197.0	87.6	85.2	58.2	1,176.0	202.8	95.1	92.7	65.9	1,171.7
jsoup	3,449.1	1,995.3	1,995.3	1,927.3	11,985.8	3,673.4	2,510.2	2,510.2	2,365.0	11,200.5
logback	1,641.7	427.6	438.0	196.0	6,592.4	1,641.3	427.6	438.0	196.0	6,591.8
ninja	585.6	227.3	180.5	159.6	1,408.6	585.6	227.3	180.5	159.6	1,408.6
okhttp	100.7	49.5	49.5	21.1	394.9	100.7	49.5	49.5	21.1	394.9
raml-java-parser	2,232.0	853.6	853.6	154.5	16,420.7	2,232.0	853.6	853.6	154.5	16,420.7
retrofit	247.1	50.6	50.9	38.0	1,308.2	247.1	50.6	50.9	38.0	1,308.2
togglz	471.8	356.1	356.3	338.2	837.9	473.5	358.8	359.0	343.0	834.0
undertow	4,677.7	2,421.9	2,406.8	1,761.6	13,307.4	4,675.9	2,435.7	2,420.6	1,775.5	13,267.8
wro4j	2,723.5	1,226.7	1,201.0	606.8	9,318.0	2,726.7	1,228.3	1,202.6	606.8	9,311.5
30 Total	84,013.0	51,821.8	51,804.9	42,333.4	299,203.5	90,479.0	60,810.3	60,793.3	52,014.6	284,820.7

5 DISCUSSION

From our process of improving PIT to track if mutants are properly covered by tests, We find that it is incredibly important to *precisely* calculate coverage in order to target tests to mutants. We find that PIT's original basic block-based coverage collection optimistically assumes that the code never throws exceptions, and hence, could yield incorrect coverage results, and in turn, incorrect assumptions about which tests should target which mutants. Moreover, if PIT finds that a test covers a line of code, it assumes that the test could cover a mutant related to any position on that line, even if the test does not execute all instructions on that line. We discussed these issues with PIT's maintainer, who was concerned that precisely collecting coverage would result in a significant performance hit. We significantly rewrote PIT's coverage collection code to improve its performance (while also correcting these bugs). With our improvements, PIT is able to precisely collect instruction-level coverage, yet still run 41% faster than the baseline version of PIT without our modifications. Our performance-enhancing changes have already been merged into PIT, with our changes for collecting instruction-level coverage still pending. We look forward to also contributing our flakiness-reducing patches back to the open-source community.

Our results are conservative, providing a lower-bound on the impact of flakiness on mutation, and may be understating the effect. Our methodology requires flaky tests to demonstrate differing coverage on repeated execution, yet some flaky tests may continue to show the same coverage even on our repeated executions, only to appear flaky another day. As research on flaky tests finds new and better ways to detect them, mutation testing can benefit as well.

Threats to Validity. There are several potential threats to generalizing our results. The projects that we evaluate may not be representative. To alleviate this threat, we select a large number of projects from different domains that have been studied by prior work. Within our dataset, we find that several effects did not generalize to all projects, for instance, we observed no flakiness in the *cors-filter* project. We have structured our findings to make clear what parts may be less broadly applicable.

The tools that we use may have bugs that propagate to our results. We attempt to mitigate this concern by using a popular open-source tool, PIT, and exhaustively examining our results for consistency, while finding, reporting and patching several bugs in PIT in the process. Given that our goal is to study non-deterministic behavior, underlying non-determinism in the systems we evaluate may leak into our findings. To ensure fair comparison, we perform all comparative analyses off of a single set of reference executions (of statement and mutation coverage), rather than repeatedly gathering coverage and mutation scores for each experiment. To support reproducibility, we make our dataset and tool publicly available [37]. Our evaluation of mutant prioritization is based entirely on simulation — the actual time needed to perform the testing may differ from what we calculate. However, it is necessary to simulate these executions to ensure that for each run, each test behaves identically, covering exactly the same statements and killing exactly the same mutants. Again, we simulate executions due to the (observed) non-determinism in the subjects that we evaluate.

6 RELATED WORK

Mutation Testing. Mutation testing is a widely studied area of research for many years; Jia and Harman present a survey on mutation testing [19]. Researchers and developers have over the years developed many mutation testing tools for different languages. Just for Java, there are several mutation testing tools including Javalanche [35], Judy [28], MuJava [27], Major [20], and PIT [7, 8]. We use and enhance PIT as part of this work. In addition to developing tools, researchers have spent many years improving mutation testing in general, such as through improving mutation operators [44] or speeding up mutation testing. For speeding up mutation testing, there was work in identifying equivalent and duplicated mutants, which artificially bloat mutation scores and are unnecessary to run [14, 32, 36]. There was also work in selective mutation testing, selecting a subset of mutants for faster evaluation [2, 13, 46–48]. In particular, Zhang et al. [47] investigated prioritizing tests to run on mutants to more quickly determine if a mutant is killed. As part of our work, we also consider prioritizing tests. However, we consider that tests can be flaky and incorporate rerunning tests until the mutant status is reliably determined. Furthermore, we measure time of running tests on mutants instead of measuring the number of test executions.

Much research in software testing has relied on mutation testing for evaluation or guidance. Fraser and Zeller used mutation testing to improve test generation [10]. Shi et al. studied the effects of test-suite reduction using mutation testing [38, 40, 41]. Mutations are also commonly used as a proxy for faults in research on test-case prioritization [9, 23, 24, 26]. All this previous work implicitly assumed tests are not flaky and did not consider their effects on the mutation testing results. In this work, we investigate how much an effect test flakiness has on mutation testing.

Flaky Tests. Luo et al. conducted an extensive study of flaky tests, studying over 200 flaky tests and categorizing them, creating a taxonomy of the root causes of flaky tests [25]. Our lightweight instrumentation that records simply if each mutation is covered is similar in spirit to DeFlaker's approach to detecting flaky regression tests [6]. DeFlaker automatically detects flaky regression tests by combining code coverage and code change information, labeling tests as flaky if the test did not execute any changed code, but had its outcome (passing/failing) change [6]. We used the same evaluation subjects as in the DeFlaker work, allowing us to have a reasonable sense of a ground-truth number of flaky tests in the subjects. Palomba and Zaidman [45] studied the co-occurrence of flaky tests and test smells, building a dataset of flaky tests by repeatedly rerunning tests and observing their status changing. Vahabzadeh et al. [43] studied bugs in test code in Apache software, categorizing 21% of the bugs as due to flaky tests. Both Gao et al. [12] and Marinescu et al. [29] found tests to have non-deterministic coverage when rerun on the same code version. Hilton et al. [17] also found non-determinism in coverage, except they reran tests as code evolved over time. Like these studies, we find non-determinism in coverage for tests, though all in the same version, which we later use to explore that effect on mutant generation and mutant execution. Test-order dependencies are one cause of flaky tests, which are often mitigated through test isolation [49]. Several techniques have been proposed to detect test order dependencies [5, 15, 18, 49] and to isolate tests to reduce their impact [4]. Lam et al. [22] published a dataset of flaky tests, where the flaky tests are categorized as order-dependent or non-order-dependent. It would be interesting future work to apply techniques from test-order dependency research in mutation testing.

Steimann et al. [42] studied fault-localization techniques and found their performance to be sensitive to non-determinism in coverage due to flaky tests. Martinez et al. [30] studied the use of automatic repair techniques using the Defects4J dataset [21] and found that flaky tests in the dataset caused techniques to give incorrect results. In our work, we specifically study the effects of flaky tests on another common area of software testing research, mutation testing, and we develop techniques to mitigate the effects of flaky tests on mutation testing.

7 CONCLUSION

Mutation testing relies on deterministic test behavior (modulo each mutant) to determine whether a mutant is killed. We find that even when tests consistently pass, they can still exhibit fine-grained non-determinism at the level of code coverage, resulting in tests that non-deterministically execute mutated code. We find that these factors can account for (on average) a four percentage point variation in overall mutation score not currently detected by existing mutation testing tools. Moreover, we find that for researchers interested in studying full mutation matrices, on average 9% of mutant-test pairs have an unknown status. We present techniques to detect and manage this flakiness in mutation testing through improved coverage collection to ensure a more reliable mapping from tests to mutants they cover and to rerun tests on mutants when they do not cover them until we obtain a reliable result. Our techniques reduce the percentage of unknown mutants by 79.4%.

ACKNOWLEDGMENTS

We thank Angello Astorga and Owolabi Legunsen for their discussions about flaky tests and mutation testing. This work was partially supported by National Science Foundation grants CCF-1421503, CNS-1646305, CNS-1740916, CCF-1763788, CCF-1763822, and OAC-1839010. We acknowledge support for research on flaky tests and test quality from Facebook, Google, and Huawei.

REFERENCES

[1] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. 2017. Applying mutation analysis on kernel test suites: an experience report. In *ICSTW*.

[2] Paul Ammann, Marcio E. Delamaro, and Jeff Offutt. 2014. Establishing theoretical minimal sets of mutants. In *ICST*.

[3] Apache Software Foundation. [n.d.]. SUREFIRE-1516. https://issues.apache.org/jira/browse/SUREFIRE-1516.

[4] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*.

[5] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE*.

[6] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: automatically detecting flaky tests. In *ICSE*.

[7] Henry Coles. [n.d.]. Real world mutation testing. http://pitest.org.

[8] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *ISSTA*.

[9] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *TSE* 32, 9 (2006).

[10] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *ISSTA*.

[11] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*.

[12] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making system user interactive tests repeatable: when and what should we control?. In *ICSE*.

[13] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. 2013. Selective mutation testing for concurrent code. In *ISSTA*.

[14] Bernhard JM Grün, David Schuler, and Andreas Zeller. 2009. The impact of equivalent mutants. In *ICSTW*.

[15] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: detecting state-polluting tests to prevent test dependency. In *ISSTA*.

[16] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In *SCAM*.

[17] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A large-scale study of test coverage evolution. In *ASE*.

[18] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*.

[19] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *TSE* 37, 5 (2011).

[20] René Just. 2014. The Major mutation framework: efficient and scalable mutation analysis for Java. In *ISSTA*.

[21] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*.

[22] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: a framework for detecting and partially classifying flaky tests. In *ICST*.

[23] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *ISSRE*.

[24] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *ICSE*.

[25] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.

[26] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *FSE*.

[27] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for Java. In *ICSE*.

[28] Lech Madeyski and Norbert Radyk. 2010. Judy - a mutation testing tool for Java. *IET software* 4, 1 (2010).

[29] Paul Marinescu, Petr Hosek, and Cristian Cadar. 2014. Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In *ISSTA*.

[30] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in Java: a large-scale experiment on the Defects4J dataset. *ESE* 22, 4 (2017).

[31] John Micco. 2017. The state of continuous integration testing @Google. https://research.google.com/pubs/pub45880.html.

[32] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*.

[33] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *ICSE-SEIP*.

[34] Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *ICSTW*.

[35] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *ESEC/FSE*.

[36] David Schuler and Andreas Zeller. 2010. (Un-) Covering equivalent mutants. In *ICST*.

[37] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing (dataset and tool). (6 2019). https://doi.org/10.6084/m9.figshare.8226332

[38] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *FSE*.

[39] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*.

[40] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating test-suite reduction in real software evolution. In *ISSTA*.

[41] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *ESEC/FSE*.

[42] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*.

[43] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *ICSME*.

[44] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. 2018. Decartes: A PITest engine to detect pseudo-tested methods. In *ASE Demo*.

[45] Andy Zaidman and Fabio Palomba. 2017. Does refactoring of test smells induce fixing flaky tests?. In *ICSME*.

[46] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: better together. In *ASE*.

[47] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*.

[48] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2012. Regression mutation testing. In *ISSTA*.

[49] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivaç Muşlu, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*.