A Study of Flaky Failure De-Duplication to Identify Unreliably Killed Mutants

Abdulrahman Alshammari
George Mason University
Fairfax, United States
aalsha2@gmu.edu

Paul Ammann
George Mason University
Fairfax, United States
pammann@gmu.edu

Michael Hilton
Carnegie Mellon University
Pittsburgh, United States
mhilton@cmu.edu

Jonathan Bell
Northeastern University
Boston, United States
j.bell@northeastern.edu

***Abstract*—Mutation testing is a technique for evaluating the efficacy of a test suite in identifying faults. However, non-deterministic behavior, particularly flaky tests, reduces confidence in mutation testing outcomes. Flaky tests introduce uncertainty in linking the cause behind killed mutants, which affects the trust in the assessment. When a flaky test "kills" a mutant, does it *reliably* kill it, or does it only *incidentally* kill it due to flakiness? This study is the first to directly examine the impact of flaky tests on killing mutants, underscoring how flaky tests can yield unreliable mutation results. Our analysis of 22 Java projects, previously examined for test flakiness, reveals that 19% of mutants killed by these flaky tests result from the flakiness introduced by at least one test. We examined the efficacy of failure de-duplication approaches in distinguishing mutants that were *reliably* killed from those that were not. We show that this approach effectively approximates the true number of mutants reliably killed, but with far less computational cost than re-running each test for each mutant.**

I. Introduction

Mutation testing has been widely used in software testing research and practice as a method to evaluate the quality of test suites [1][2]. Mutation testing measures test adequacy by introducing small changes into the code and checking how often tests observe these changes. The idea is to make these changes *automatically* by applying mutation operators that create variants of the original code: "mutants". Then, the test suite is run against each of these mutants. A mutant is "killed" if all the tests pass on the original code but at least one of these tests fails on the mutant; otherwise, it is called "survived." Each mutant that survives is either an instance of a possible bug that should be detected by the test suite or an "equivalent mutant" — one that is semantically equivalent to the original program. To evaluate the test suite, the mutation score is computed, representing the portion of the killed mutants to all generated mutants. Mutation scores are also used in research to evaluate testing techniques [2].

However, the outcome of mutation testing is reliable only if the tests are deterministic, i.e., they are always consistent with the same outcome. In the context of mutation, a mutant is killed if and only if a test detects the behavioral change induced by the mutation. However, tests could behave non-deterministically, i.e., pass and fail, even running on the same code [3]. Non-deterministic tests, known as flaky tests, have been widely discussed [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. Flaky tests breaks the trust of the testing outcome and may cause unreliable evaluation in terms of the mutation testing.

Ideally, flaky tests would be detected and removed from test suites (or fixed to no longer be flaky). However, in practice, flaky tests remain in test suites even after being identified as flaky because they can be valuable in detecting actual defects. Hence, flaky tests in the test suite are another significant concern for mutation testing. Since flaky tests can both pass and fail, they could also have two different (unreliable) outcomes for a mutant: *survived* and *killed*. Mutants that survive may indicate the need to create additional test cases, while a killed mutant raises confidence in the effectiveness of the test cases detecting the mutants. In both scenarios, unreliably killed mutants (which sometimes are killed and survived) can lead to a misinterpretation of the mutation testing outcome.

Studies have shown the impacts of flaky tests on mutation testing, particularly highlighting how mutation scores can vary when considering unreliably killed mutants. Shi et al. show that the mutation scores of their experiment vary by four percentage points due to flaky tests [2]. Mutants killed by flaky tests cannot be accurately assessed, leading to mutation scores that may not truly reflect the actual mutation analysis. Furthermore, it may require additional time to definitively determine if a specific mutant is detected, considering the possibility that the mutant was killed unreliably by flaky tests.

When developers utilize mutation testing, they might want to know if a mutant was reliably killed. The state-of-the-art approach for determining whether or not a mutant is reliably killed is simply to rerun the entire mutation analysis process multiple times. However, this approach is time-consuming, and as far as we know, no study has measured how many mutants are unreliably killed. This article describes an empirical study of how often flaky tests unreliably kill mutants. Drawing on our prior research and dataset that evaluated the efficacy of failure de-duplication in identifying flaky test failures, we study the efficacy of this promising approach to determine whether mutants are reliably killed [20], [21]. Examining the outcomes of 20 repetitions of mutation analysis on flaky tests of 22 Java projects, we find that 19% of killed mutants are, in fact, *unreliably killed*. We also found that our previous failure de-duplication approach effectively detected unreliably killed mutants. The approach detected 2,950 (out of 3,237 unreliably killed mutants via rerun) with varied over-classification rates

(labeling reliably killed mutants as unreliable) across the studied projects. We show the result of our study in Section III.

II. Study

Unreliably killed mutants (killed by a flaky test in one run and survived by the same test in another) reduce trust in mutation analysis results. This implies uncertainty each time a flaky test kills a mutant, leading to questions about whether or not the mutant was reliably killed. Our goal is to increase developers' confidence in the validity of killed mutants, ensuring that these killed mutants result from the introduced changes in the code rather than being influenced by flakiness causes. Our study begins by showing how often a flaky test could have unreliably killed mutants. Then, following our prior work [20], we investigate the feasibility of using a lightweight failure de-duplication-based approach to determine whether a given mutant is unreliably killed.

A flaky test may fail for various reasons, including identifying a fault. Hence, in mutation testing, flaky tests might appear to kill a mutant for various reasons, including flakiness or because they genuinely detect a bug introduced by the mutant. Identifying the reasons behind test failures in mutation testing is important for counting the number of reliably killed mutants and obtaining more accurate mutation scores.

By relying on the definition of a flaky test failure, we can construct a procedure to determine which mutants are reliably killed and which are not. Specifically, since flaky test failures occur non-deterministically, a killed mutant can be confirmed as unreliable by *rerunning* the test multiple times. If different outcomes (passing and failing) are observed, the killed mutant is confirmed as *unreliable*. However, rerunning tests is a resource-intensive process, and it becomes even more costly when running the entire test suite against *all* mutants. As mutation testing itself could be expensive at some scales of projects, determining flaky mutants should not introduce an additional overhead cost.

Our previous work discusses the ability to distinguish between flaky and actual failures of flaky tests using failure logs [20]. We compare new failures with prior flaky failures using the text-matching de-duplication approach (comparing stack traces to find matches between two given failures). We found that flaky failures are repetitive, meaning that a test, once it fails due to flakiness, is most likely seen previously if the test has a prior history of flaky failures.

In our prior work detecting flaky failures [20], we compare test failures with previous flaky and true failures (with the same code under test revision). We have not explored whether the proposed approaches could aid mutation testing by detecting flaky tests that kill mutants unreliably. The result of our prior work encourages us to investigate the ability of our approaches to detect unreliably killed mutants by using the original failures of the flaky tests.

A. Study Design

Previous studies have explored several de-duplication-based approaches that have been used in distinguishing between flaky and true failures, as detailed in [22] and our study [20]. The core idea behind these approaches is that flaky failures might have different attributes than true failures, and hence, an approach that can match similar failures to each other may also be able to distinguish flaky from true failures. Our prior work compared failure de-duplication approaches based on machine learning and simple text matching [20]. In this work, we examine only the text-matching approach, as our prior work identified it as effective and easy to deploy. Details about the text-matching approach are described in our prior work [20], and we reuse the implementation from our prior artifact [23]. We use these scripts to attempt to match the failure logs from killed mutants against known flaky failures.

We conduct our study using our previously published dataset [21] that contains both: 1) confirmed flaky failure logs from flaky tests (drawn from the FlakeFlagger dataset [5]), and 2) 20 sets of mutation results of these flaky tests. However, this dataset *only* contains mutation analysis results for tests previously found to be flaky in these projects and does *not* contain mutation analysis results for the entire test suite. Hence, our study design will not support conclusions regarding an overall impact on a test suite's mutation score. Instead, we focus our research on understanding the efficacy of a lightweight approach to determine whether a mutation result should be trusted or potentially ignored due to flakiness. Further details regarding the limitations of this study are discussed in Section IV.

To evaluate the performance of the approach, we compare the approach findings against the baseline approach *rerun*, which is also provided in our prior study dataset. Due to the challenge of obtaining the total number of killed mutants (given that mutation analysis is based on a subset of flaky tests), it was hard to build a confusion matrix to evaluate the approach. Instead, we assess the performance by quantitatively the number of unreliably killed mutants detected by the approach versus rerun.

III. Results

Based on the conducted study, our primary objective is to address the following main research questions:

- **RQ1: How often does a flaky test cause unreliably killed mutants?** In this **RQ**, our goal is to show how often mutants can be killed in an unreliable way and discuss if the portion of unreliably killed mutants is a significant threat to the validity of the mutation analysis.
- **RQ2: How effectively can de-duplication failure approaches detect unreliably killed mutants?** We investigate using duplication failure approaches in detecting which mutant is reliable or unreliable.

A. RQ1: How often does a flaky test cause unreliably killed mutants?

We examined the mutation dataset from our prior study [21], which includes 22 Java projects, where each confirmed flaky test was run 20 times against all mutants. The status of each

TABLE I
NUMBER OF MUTANTS AND THEIR STATUS PER PROJECT
This table displays the number of flaky tests in each project and the total number of mutants (Column Total). The columns "Killed Mutants" show the total number of any mutant that is killed at least once, followed by mutants that are *reliably* killed, (unreliably) killed, and mutants that are reliable by some tests and not others. The column "Per Test" shows how many flaky tests have never caused a mutant to be unreliably killed (never), followed by flaky tests that cause at least one unreliably killed mutant.

Project	Total		Killed Mutants				Per Test	
	Tests	Mutants	Total	Reliably	Unreliably	Both	never	[1:n)]
activiti	30	15,574	3,989	2,819	849	321	5	25
okhttp	99	5,063	2,239	1,501	483	255	19	80
alluxio	114	16,595	1,865	1,739	63	63	38	76
ambari	51	6,183	1,304	1,302	1	1	49	2
logback	20	7,304	1,194	885	243	66	13	7
wildfly	18	1,317	1,046	1,046	0	0	18	0
httpcore	22	5,482	1,003	960	32	11	6	16
hbase	22	22,056	749	518	231	0	19	3
io-undertow	7	16,305	716	707	6	3	5	2
java-webSocket	23	1,476	516	137	278	101	0	23
wro4j	14	3,662	423	414	7	2	12	2
spring-boot	12	7,929	422	421	1	0	11	1
hector	33	1,560	359	358	1	0	32	1
orbit	7	1,183	311	175	107	29	0	7
handlebars.java	1	1,842	182	147	35	0	0	1
achilles	2	2,701	154	154	0	0	2	0
ninja	1	1,603	147	120	27	0	0	1
http-request	18	468	109	109	0	0	18	0
zxing	1	12,499	84	76	8	0	0	1
elastic-job-lite	3	1,726	66	66	0	0	3	0
commons-exec	1	472	60	59	1	0	0	1
assertj-core	1	4,376	30	18	12	0	0	1
22 Java Projects	500	137,376	16,968	13,731	2,385	852	250	250

mutant was recorded based on these 20 runs. Our follow-up analysis involved processing the mutation reports per test to determine the number of *reliably* and *unreliably* killed mutants. Table I presents the results of this analysis per project.

Taking the project *activiti* as an example from Table I, we identified 30 flaky tests that had caused at least one mutant to be killed. Out of the 15,574 mutants in this project, we found 3,989 killed mutants, which corresponds to 25% of the total number of mutants in this project. The column "Killed Mutants" shows that there are 2,819 mutants that were always reliably killed by any of the 30 tests, 849 mutants that were always unreliably killed, and 321 mutants that were reliably killed by some tests and unreliably by others. The column "Per Test" indicates how many tests out of 30 were never responsible for causing a mutant to be unreliably killed (never) and were only responsible for at least one mutant being unreliably killed ([1:n)).

Overall, we identified 3,237 *unreliably* killed mutants (the sum of the columns "Unreliably" and "Both") and 13,731 *reliably* killed mutants out of a total of 137,376 mutants across the 22 Java projects. This means that approximately 19% of the mutants killed by at least one flaky test were unreliably killed. It is challenging to assert how this might impact the overall mutation analysis, as we only consider mutation results for known flaky tests. However, the count of flaky mutants could be considered a minimum number based on the reported result per project, assuming a mutant is unreliably killed if it was killed *and* survived by at least one test.

It is important to emphasize that half of the 500 studied flaky tests are linked to causing at least one mutant to be unreliably killed. This implies that half of the flaky tests reported in [5] could not be replicated in our studied dataset [21]. Flaky tests generally have reproducibility challenges, especially when executed across different run environments. Hence, these factors may influence some flaky tests, leading to not capturing *all* flaky tests in the studied dataset. We further analyze flaky tests that have not resulted in any mutant being unreliably killed. In *wildfly*, where no unreliably killed mutants were observed, all the flaky tests were initially flaky, sharing the same exception: *RuntimeException*. In other projects, out of 33 reported flaky tests in *hector*, 32 were flaky due to *HCassandraInternalException*, and none of these flaky tests were responsible for any of the unreliably killed mutants. Similarly, in *ambari*, 47 flaky tests exhibited flakiness due to *ProvisionException*, and none of them were responsible for any of the unreliably killed mutants.

The percentage of unreliably killed mutants compared to reliably killed mutants showed variation across different projects. Four projects, with a total of 41 flaky tests, had *zero* unreliably killed mutants. On the other hand, seven other projects demonstrated that at least 25% of the killed mutants were labeled as unreliably killed. Our findings revealed that a significant portion, around 36%, of the total number of unreliably killed mutants originated from a single project named *activiti*. In the case of *java-websocket*, there were more unreliably killed mutants than reliable ones, and every studied flaky test caused at least one mutant to be unreliably killed. It is challenging to draw a conclusive percentage for unreliably killed mutants in a single project, as this metric can be influenced by various factors such as the run environment.

The columns "Unreliably" and "Both" present the counts of unreliably killed mutants from two perspectives: 1) killed mutants that any test has reliably killed, and 2) killed mutants that were at least reliably killed by at least one flaky test. While the number of mutants marked exclusively as unreliable (column "Unreliably") is higher than "Both," it is important to recognize that this ratio might change when also considering mutation results of non-flaky tests. In the context of this study, our idea is to show that developers should consider that the presence of flaky tests in mutation analysis could result in unreliably killed mutants by at least one test.

Summary. We observed that about half of the flaky tests were responsible for unreliably killed mutants, constituting 19% of the killed mutants across the 22 Java projects. The ratio of unreliably killed mutants could vary from one project to another. Hence, this highlights the significance of identifying which killed mutant was reliably killed and which is unreliably killed for achieving *better* mutation analysis.

B. RQ2: How effectively can de-duplication failure approaches detect unreliably killed mutants?

Table II shows the performance of the de-duplication approach (text-based matching [20]) in detecting unreliably

TABLE II
THE EFFICACY OF DETECTING FLAKY MUTANTS.
"Rerun" columns show the number of *reliably* and *unreliably* killed mutants after running the tests 20 times. "De-duplication" shows how many mutants are determined to be reliably or Unreliably killed, based on the de-duplication approach described in [20]. Finally, "De-duplication Performance Against Rerun" shows the efficacy of the de-duplication approach compared to the rerun ground truth.

	Total			Rerun		Deduplication		Deduplication Performance Against Rerun
Project	**Test**	**Mutants**	**Killed**	**Reliably**	**Unreliably**	**Reliably**	**Unreliably**	Only Rerun Rerun & Deduplication Only Deduplication
activiti	30	15,574	3,989	2,819	1,170	2,162	1,827	
okhttp	99	5,063	2,239	1,501	738	1,549	690	
java-webSocket	23	1,476	516	137	379	14	502	
logback	20	7,304	1,194	885	309	676	518	
hbase	22	22,056	749	518	231	379	370	
orbit	7	1,183	311	175	136	156	155	
alluxio	114	16,595	1,865	1,739	126	1,536	329	
httpcore	22	5,482	1,003	960	43	772	231	
handlebars.java	1	1,842	182	147	35	131	51	
ninja	1	1,603	147	120	27	112	35	
assertj-core	1	4,376	30	18	12	30	0	
wro4j	14	3,662	423	414	9	406	17	
io-undertow	7	16,305	716	707	9	288	428	
zxing	1	12,499	84	76	8	80	4	
ambari	51	6,183	1,304	1,302	2	1,298	6	
commons-exec	1	472	60	59	1	57	3	
spring-boot	12	7,929	422	421	1	422	0	
hector	33	1,560	359	358	1	347	12	
achilles	2	2,701	154	154	0	148	6	No Detected Flaky Mutants
wildfly	18	1,317	1,046	1,046	0	1,046	0	No Detected Flaky Mutants
elastic-job-lite	3	1,726	66	66	0	66	0	No Detected Flaky Mutants
http-request	18	468	109	109	0	102	7	No Detected Flaky Mutants
22 Total Projects	500	137,376	16,968	13,731	3,237	11,777	5,191	

killed mutants compared to rerun results. Using *activiti* as an example, 30 flaky tests were run against 15,574 mutants, resulting in 3,989 killed mutants. Of these, 1,170 were labeled as unreliably killed by at *least* one flaky test using rerun (with 2,819 labeled as reliable). The de-duplication approach identified 1,827 killed mutants as unreliable and 2,162 as reliable. In the column "Deduplication [22], [20] Performance Against Rerun," the bar plot uses three colors: *orange* bar shows unreliably killed mutants confirmed by rerun but not detected as unreliably killed by the approach. The *green* bar represents unreliably killed mutants confirmed by both rerun and the approach. In contrast, the *red* bar indicates mutants detected as unreliably killed by the approach but not reported as unreliably killed by rerun.

When comparing the outcomes of the de-duplication approach with those of rerun, we examine both false positives (labeled as unreliably killed but were reliably killed) and false negatives (marked as reliably killed but were unreliably killed). The stacked bar charts in Table II show false positives in red and false negatives in orange. Taking the *wro4j* results as an example, out of 423 total mutants killed, nine were confirmed as unreliable using rerun (which provides our ground truth). By the de-duplication approach, 17 killed mutants (out of 423) were categorized as unreliable. Of the 17 labeled as unreliable by the matching approach, 9 are true positives (green bar). However, the remaining eight mutants initially classified as unreliable were found to be reliable by rerun. Considering the approach in *wro4j*, the portion of unreliably killed mutants can vary by as much as 17, indicating the lowest anticipated ratio of reliable mutants.

The rerun approach (our ground truth) identified 3,237 unreliably killed mutants, as shown in the bottom row of Table II. Among these, 2,950 unreliable mutants were also detected as unreliable by the de-duplication approach, while 287 unreliably killed mutants were misclassified as reliable. The approach misclassified 2,241 reliable killed mutants as unreliable, resulting in 5,191 unreliable mutants by the de-duplication approach. The ratio of the unreliably killed mutants identified by the approach to the total number of killed mutants may differ if the total number of tests represents the entire test suite and not just flaky tests. Hence, we could not accurately assess the approach's accuracy (further details in Section IV).

Overall, the approach was able to identify 91% of the confirmed unreliably killed mutants (2,950 out of 3,237) while mislabeling 287 killed mutants. The approach misclassifies mutants as unreliable and killed in most projects at different rates (represented by the red bar). The approach did not detect unreliably killed mutants in *assertj-core* and *spring-boot*. Among the studied projects, 7 had at least 100 unreliably killed mutants. The de-duplication approach effectively detected the majority of unreliably killed mutants in these projects (2,839 out of 3,089). However, *alluxio* and *okhttp* have a higher number of misclassified unreliably killed mutants, as indicated by the orange bar.

We found some of the unreliably killed mutant failures not previously detected in test flaky failures, represented by the orange bar in each project. For instance, in *alluxio*, the exception *TTransportException* occurred 12 times (out of 55 unreliably killed mutants that the approach could not detect). In *okhttp*, one exception named *NoSuchFieldError* appeared in

60% of the unreliably killed mutants that exclusively occurred in unreliably killed mutants. In the case of *assertj-core*, we identified only one flaky failure (from a single flaky test) that did not match any of the killed mutants, resulting in zero unreliably killed mutants detected by the approach.

The approach appeared to over-misclassify mutants as unreliable in four projects, such as *httpcore* (231 VS 43) and *undertow* (428 VS 9). With closer inspection of these projects, we found that the majority of these misclassified mutants fail due to *AssertionError* as a failure exception (75% of unreliably killed mutants in *httpcore* and 60% in *undertow*). In our prior work that aimed to separate true failures from flaky failures, we also found that failures involving an *AssertionError* were likely to be misclassified [20].

The ratio of falsely labeled mutants as unreliably killed by the approach to the total number of reliably killed mutants is relatively low in most projects. However, in specific projects like *undertow*, the ratio exceeds 50%. This ratio is crucial for developers to consider when deciding whether to utilize the approach. The false positive rate, corresponding to the red bar in Table II, can establish a lower boundary for the potentially killed mutants to be considered unreliable. This boundary enhances confidence in the accuracy of the total number of *reliable* killed mutants.

As discussed earlier, rerunning the tests on *every* mutant is extremely expensive. On the other hand, the cost of the approach to detect unreliably killed mutants relies on leveraging the prior failures of the flaky tests. The main drawback of the approach is that it lacks historical flaky failure context. The approach might be considered as a metric to organize rerunning test mutant pair by *prioritizing* running tests that have more unreliably killed mutants by the approach.

Summary. We found that the approach was effectively able to determine at least 91% of the unreliably killed mutants. The findings in most projects suggest that developers could consider that the unreliably killed mutants can be detected using the text-matching approach.

IV. Discussion and Threats to Validity

Our study design is subject to several limitations that threaten the validity of our conclusions.

Collecting a large, ground-truth dataset of unreliably killed mutants is highly time-consuming, requiring repeatedly collecting mutation analysis for a project. The dataset we utilized is from our prior study [20], where mutation analysis served as a stand-in for actual failures, and de-duplication-based approaches were used to detect flaky failures. During our analysis in the prior work, we excluded any mutant that was not reliably killed. This work includes these unreliably killed mutants to see how the deduplication approach could detect them. However, this dataset only includes the mutation analysis results on known flaky tests. It does not provide a complete mutation analysis for each test suite, so the performance of the deduplication approach may be different on other tests within the same test suites. Without mutation results for all tests, it is not possible for us to measure the mutation score directly or to measure the impact of flaky tests on the mutation score. However, the goal of our study was not to evaluate the impact of flaky tests on mutation scores (as Shi et al [2]), but to study the efficacy of failure deduplication in determining which killed mutants were unreliably killed.

More reruns of more tests and mutants will undoubtedly result in a larger dataset, and we believe this will be the basis of future work in flaky tests and mutation analysis. A more rigorous evaluation would consider all tests and report additional metrics, such as a complete confusion matrix. We explicitly avoid reporting a confusion matrix for each project to underscore the sensitivity of our results to the limited set of tests studied and call for future research to study these methods in other contexts.

We studied our previously proposed failure deduplication approach [20] and inherited the same limitations of that underlying approach. For example, different test suites may have various failure symptoms, making the approach more or less effective. Failure logs might also be merged from various runtime environments, complicating the analysis. Our results may or may not generalize to other projects.

Test-mutant pair evaluation could be more informative in mutation analysis. The deduplication approach we use in our study labels each mutant as an unreliably killed mutant if it is unreliably killed by at least one flaky test. It is important to mention that some unreliably killed mutants are reliably killed by other tests, as shown in the column "Both" in Table I. The number of times a mutant is unreliably killed could assist the researchers and developers to *prioritize* tests during mutation testing. Due to the limitation of the dataset, we consider this also a main future work to enhance the mutation analysis by considering the failure deduplication approach as a standard to evaluate test-mutant evaluation.

V. Related Work

Flaky tests have been widely discussed in software testing research community [4], [5], [6], [7], [8], [9], [10], [11], [12]. In the context of mutation analysis, there are significant research efforts about flaky tests in mutation testing [2], [24], [25], [26], [27]. Du et al. studied the reasons behind test failures contributing to mutation kills in mutation testing [24]. While they do not explicitly include flakiness as a cause, they acknowledge that flaky tests may exist in the projects they analyze in their experiment. They view this as a significant threat to the validity of their results. Vercacmmen et al. delve into the integration of mutation testing with integration testing and emphasize that flaky tests are a significant concern that needs to be addressed [25].

Shi et al. discuss the effect of having flaky tests in the test suite during mutation testing [2]. Shi et al. focused on non-deterministic test coverage in their experiment with 30 Java projects. They found that 22% of statements exhibited inconsistent coverage when test suites were repeatedly executed. They observed that due to the flakiness in test coverage, about 5% of mutants had an unknown status, leading to uncertainty in the mutation score. A key limitation of this prior work is

that the authors did *not* attempt to rerun killed mutants to see whether or not they were reliably killed. Assuming all "unknown" mutants were killed, the mutation score would be 82%, and assuming they all survived, the score would be 78%. This underscores how non-deterministic coverage significantly affects the reliability of mutation scores as a measure of test effectiveness. In our case, outlined in Section IV, we encountered challenges in computing mutation scores. We also confirm unreliably killed mutants by the reported rerun result, a significantly stronger ground truth.

VI. Conclusion

We found that 3,237 of mutants killed by flaky tests in 22 open-source Java projects were *unreliably killed*, corresponding to 19% of all killed mutants. unreliably killed mutants can provide false confidence in a test suite's fault-finding ability. Recognizing the difficulties of rerunning the entire test suite for every mutant, our research demonstrates that failure de-duplication effectively identifies unreliably killed mutants. Using this approach, we found that it approximates the number of reliably killed mutants and was able to detect up to 91% of the mutants categorized as unreliable in their killing status. Within the limitation we discuss, our findings could encourage research to address this concern further for more *reliable* mutation analysis.

References

[1] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[2] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 112–122. [Online]. Available: https://doi.org/10.1145/3293882.3330568

[3] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653.

[4] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 403–413.

[5] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1572–1584.

[6] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[7] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 312–322.

[8] V. Pontillo, F. Palomba, and F. Ferrucci, "Static test flakiness prediction: How far can we go?" *Empirical Softw. Engg.*, vol. 27, no. 7, dec 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10227-1

[9] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021.

[10] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empirical Softw. Engg.*, vol. 28, no. 3, apr 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10307-w

[11] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1471–1482. [Online]. Available: https://doi.org/10.1145/3377811.3381749

[12] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "Nondex: A tool for detecting and debugging wrong assumptions on java api specifications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 993–997. [Online]. Available: https://doi.org/10.1145/2950290.2983932

[13] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, oct 2021. [Online]. Available: https://doi.org/10.1145/3476105

[14] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 244–255.

[15] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1381–1392.

[16] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, "The importance of discerning flaky from fault-triggering test failures: A case study on the chromium ci," *arXiv preprint arXiv:2302.10594*, 2023.

[17] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 857–862. [Online]. Available: https://doi.org/10.1145/3236024.3275529

[18] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

[19] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 492–502. [Online]. Available: https://doi.org/10.1145/3379597.3387482

[20] A. Alshammari, P. Ammann, M. Hilton, and J. Bell, "230,439 test failures later: An empirical evaluation of flaky failure classifiers," 2024.

[21] ——, "Flaky and True Failures Logs to Accompany "230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers"." Zenodo, Jan. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10531160

[22] G. An, J. Yoon, T. Bach, J. Hong, and S. Yoo, "Just-in-time flaky test detection via abstracted failure symptom matching," 2023.

[23] A. Alshammari, P. Ammann, M. Hilton, and J. Bell, "Failure log classifiers," https://github.com/AlshammariA/FailureLogClassifiers, 2024.

[24] H. Du, V. K. Palepu, and J. A. Jones, "To kill a mutant: An empirical study of mutation testing kills," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 715–726.

[25] S. Vercacmmen, M. Borg, and S. Demeyer, "Validation of mutation testing in the safety critical industry through a pilot study," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 334–343.

[26] S. Habchi, M. Cordy, M. Papadakis, and Y. L. Traon, "On the use of mutation in injecting test order-dependency," *arXiv preprint arXiv:2104.07441*, 2021.

[27] Y. Chen, A. Yildiz, D. Marinov, and R. Jabbarvand, "Transforming test suites into croissants," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1080–1092.