To Kill a Mutant: An Empirical Study of Mutation Testing Kills

Hang Du
University of California, Irvine
Irvine, USA
hdu5@uci.edu

Vijay Krishna Palepu
Microsoft, Silicon Valley Campus
Mountain View, USA
vijay.palepu@microsoft.com

James A. Jones
University of California, Irvine
Irvine, USA
jajones@uci.edu

ABSTRACT

Mutation testing has been used and studied for over four decades as a method to assess the strength of a test suite. This technique adds an artificial bug (*i.e.,* a mutation) to a program to produce a mutant, and the test suite is run to determine if any of its test cases are sufficient to detect this mutation (*i.e.,* kill the mutant). In this situation, a test case that fails is the one that kills the mutant. However, little is known about the nature of these kills. In this paper, we present an empirical study that investigates the nature of these kills. We seek to answer questions, such as: How are test cases failing so that they contribute to mutant kills? How many test cases fail for each killed mutant, given that only a single failure is required to kill that mutant? How do program crashes contribute to kills, and what are the origins and nature of these crashes? We found several revealing results across all subjects, including the substantial contribution of "crashes" to test failures leading to mutant kills, the existence of diverse causes for test failures even for a single mutation, and the specific types of exceptions that commonly instigate crashes. We posit that this study and its results should likely be taken into account for practitioners in their use of mutation testing and interpretation of its mutation score, and for researchers who study and leverage mutation testing in their future work.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

mutation testing, mutant detection, test failure classification, empirical study

ACM Reference Format:
Hang Du, Vijay Krishna Palepu, and James A. Jones. 2023. 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 (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598090

1 INTRODUCTION

Mutation testing was first proposed by DeMillo [9] as a means to assess the effectiveness of a test suite, and moreover to identify weaknesses that can be strengthened through additional testing efforts. To assess the effectiveness of a test suite, mutation testing repeatedly injects an artificial bug (*i.e.,* a *mutation*) into the tested software to determine if any test cases fail due to the introduced mutation. When at least one test case is sensitive enough to fail due to the mutation, we consider the mutated program (*i.e.,* the *mutant*) to be *killed*, and for that mutant, the test suite is considered to be effective. In contrast, when no test cases fail on the mutant, the mutant is said to *survive*, and this survival can be an indicator of a weakness in the test suite (except in cases of equivalent mutants that cannot be killed). In the end, a *mutation score* is computed as the percentage of killed mutants among all non-equivalent mutants that were created and tested — a higher mutation score connotes a relatively effective test suite, whereas a lower score connotes a relatively less effective suite.

Since its inception, mutation testing has been a well-researched topic (*e.g.,* [17, 19, 30, 33, 44]), been used for various purposes (*e.g.,* [14, 16, 18, 29, 30, 38, 41]), and in recent years, has been made computationally efficient enough to be useful in real-world development and testing (*e.g.,* [3, 8, 32]). However, despite all of these studies, researchers and practitioners currently have an incomplete understanding of how mutation testing functions in practice, particularly with regard to how mutants are killed.

Mutants can be killed in multiple ways. The most obvious way is for a test case to fail its oracle. However, a mutant can also cause a crash of the program, and in some cases, such crashes are guaranteed if the mutation is executed (consider the situation when the mutation is setting a variable's value to 0 which is the denominator of a division computation). Moreover, sometimes the test will fail due to an assertion placed within the source code (*e.g.,* defensive programming practices [34]). Regardless of how the test suite fails, the mutation is considered to be killed, and thus, the mutation score reflects the increased effectiveness of the suite, which could be misleading if the test suite, itself, is not the source of the failure.

Some researchers have identified that some mutations are more *trivial* than others (*e.g.,* [2, 5, 13, 20, 24]) or that some mutations always cause crashes when executed (*e.g.,* [20, 26]). Most of these observations were made during experimentation, and as such, they were merely noted in their experimental designs to document how they addressed such issues.

Some research efforts have directly studied how mutation testing is affected by various types of killing behavior. For example, Just *et al.* [20] deem a mutant as *trivial* if all test cases that execute its mutation lead to an exception. However, in this past work, no distinction was made between exceptions that were thrown by the system (*e.g.,* divide-by-zero) versus those thrown by a developer-created, defensive-programming assertion in the code. In the latter case of the exception thrown by defensive programming, one possible

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598090

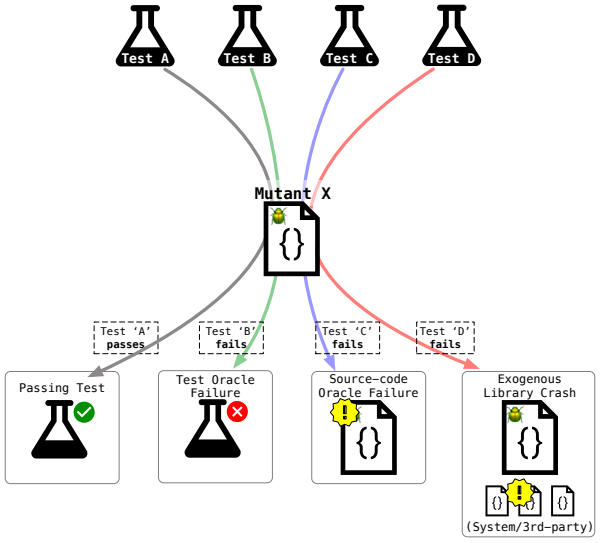

Figure 1: Possible execution outcomes when multiple tests execute a single mutant.

interpretation is that the testing is actually effective and desirable, insofar as the test provides the necessary conditions to execute the mutation, and the thorough checking of the state was performed by the assertion in the code — in such case, perhaps the developers should be recognized for effective defensive programming, and perhaps also the mutation score should reflect that, rather than considering the mutant as trivial and undesirable.

Such diversity of mutation-killing behavior can (and often *should*) be taken into account when using mutation testing. For example, mutation scores may mislead developers as to the effectiveness of the code under test. Or, developers may be encouraged to write more test cases, when perhaps source-code-based defensive programming practices are effective at catching bugs. Another example is research techniques that use mutation testing as a dependent technique (*e.g.*, fault localization [18, 25, 29]), which may be affected by ways in which tests fail.

To better understand these issues and thus to better inform future mutation-testing tool implementations, mutation-score metrics, and research endeavors, we conducted an empirical study of how mutations are killed in mutation testing. This work presents the first focused study, specifically into how mutants are killed, in practice. We provide a taxonomy of test-case failures as a result of mutations, and we further develop an operationalization for how to classify actual mutation-testing failures. To enable our empirical investigation of mutation kills, we implemented an experimental framework that involves extending a popular and well-known mutation testing tool and source-code implementation. We studied ten popular and widely used open-source systems, performed mutation testing on over 50,000 mutants, executed over 2.5 million test runs, and analyzed the specific actions of each test case that resulted in a mutant kill.

We found a number of startling results that should give concern to practitioners' interpretations of mutation scores, as well as researchers' use of mutation testing for other automated techniques, or at least, these results should be taken into consideration in such

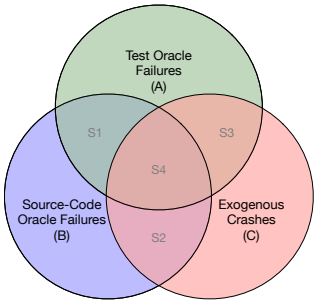

Figure 2: Taxonomy of mutation killing reasons. Mutants can be killed due to test-oracles failures, source-code oracles failures, exogenous crashes, or any combination thereof.

future work. We found that crashes can cause as much as 46.2% of failed test runs and as much as 43.8% of killed mutants (and thus the mutation score). Moreover, regardless of the software system, such crashes substantially contributed to the number of failures (and kills). Additionally, test oracles (which are the traditionally assumed cause of test failures and mutation kills) contributed as little as 11.8% of test failures for one subject. All 10 subjects showed mutants that were killed due to all failure causes from our taxonomy, with one subject exhibiting as much as 21.3% of all killed mutants failing due to all of the failure causes (*i.e.,* the test cases that failed for that mutant failed due to each of the failure causes). Moreover, we report additional findings on the exception types that cause crashes and failures.

The contribution of this paper can be summarized as follows:

- We propose a mutation test-failure taxonomy that categorizes the ways in which test cases can cause mutation kills.
- We implement our classification technique with custom dynamic instrumentation and customized extensions to a mutation-testing framework. Our implementation, experimental setup, and experimental data are open source[1] and available for replication [10].
- We studied 10 popular open-source projects, performing mutation testing on over 50,000 mutants, and executing over 2.5 million test runs. We found several revealing results across all subjects, including that: (1) crashes contribute substantially to test failures that kill mutants, (2) test-failure causes are often diverse even for a single mutation, and (3) crashes are caused by a variety of exception types.

2 BACKGROUND AND MOTIVATING EXAMPLE

In this section, we introduce a taxonomy designed to categorize the causes of test failures. We elucidate the variety of killing causes within our taxonomy using a minimalist code example, demonstrating how a single mutant can be eliminated in several ways. Moreover, we discuss prior studies that have explored various definitions of mutant stubbornness and triviality.

[1] https://github.com/spideruci/MutationKills

2.1 Mutation Test-Failure Categorization

For the purposes of mutation testing, there are two main outcomes when a single test executes a specific mutant: (1) the mutant survives if the test and all previously executed tests covering the mutation pass; or (2) the mutant is killed because the test fails. However, as we noted, a mutant kill can be caused by multiple types of test failures. How a mutant dies in a test run becomes relevant, particularly when considering the motivations for the use of mutation testing. For instance, when using mutation testing for its traditional purpose of assessing a test suite and strengthening it, we may prefer to have more test cases fail due to the tests' oracle (*i.e.,* testing whether the actual output/behavior matches the expected output/behavior) rather than failing due to a crash. In contrast, if employing mutation testing to replicate crashes that real users encounter, it may be more useful to have mutants that die by crashes (*e.g.,* via uncaught runtime exceptions).

We classify test-case failures encountered during mutation testing into three main categories: (1) *test oracles*, (2) *source-code oracles*, and (3) *crashes*. In the first two categories of test oracles and source-code oracles, test-case failures originate from within the program and its test suite, and are considered *endogenous* failures. These endogenous failures include test cases that fail in the traditional way of failing its test oracle (*e.g.,* JUnit's `assertEquals()` method) and also include failures that are caused by the developers' defensive programming (*e.g.,* in-code pre/post-condition checks, boundary-condition checks, `java.lang.Assert` calls). In both cases, the failure modes were anticipated by the developers or testers. Note that traditional mutation testing only recognizes test oracles as a source for test failures—perhaps understandably so, given its intention of assessing *test* effectiveness.

The third main category of test failures are those that originate outside the first-party source code (*i.e.,* exogenous failures), and as such are outside the purview of the developers/testers. These exogenous test failures likely could not be anticipated by the developers, and as such, may be less useful for assessing the strength of the test suite and/or error checking within the program's source code. We call this category of test-case failures *exogenous crashes* (or simply *crashes*), and these can stem from checks in the virtual machine (*e.g.,* a divide-by-zero error exhibited by `java.lang.ArithmeticException` or a VM system error, such as `java.lang.OutOfMemoryError`) or from 3rd-party code (*e.g.,* libraries, frameworks) that throws exceptions that are uncaught.

Figure 1 depicts the different execution outcomes when a mutant is executed by multiple tests — three of which are test failures with different causes: test oracles, source-code oracles, and exogenous crashes. Indeed, different mutations in a software system can be killed due to specific or a combination of such reasons, illustrated by the Venn diagram in Figure 2. Figure 2 shows a hypothetical distribution of mutants in a program, bucketed into different killing reasons, or their combinations. For instance, the S3 "slice" of the Venn diagram depicts mutations killed by both test assertions and exogenous crashes across different test runs. As such, we study both facets of mutant kills: the manner of test execution failures that kill mutants, and how the individual mutants can cause execution failures of different types.

2.2 Motivating Code Example

We now use a minimal example to better illustrate how a mutant gets killed by test oracles, source code oracles, or exogenous crashes and discuss some relevant test failure scenarios.

Listing 1: Source Code Class Example

```
1  public class CodeClass {
2
3    public int getDivisor() {
4      return 3;
5      // mutation1: mutate the return value to 0
6    }
7
8    public int getDividend() {
9      return 0;
10   }
11
12   public int div1() {
13     int divisor = getDivisor();
14     int dividend = getDividend();
15     assert divisor != 0;
16     return dividend/divisor;
17   }
18
19   public int div2() {
20     int divisor = getDivisor();
21     int dividend = getDividend();
22     return dividend/divisor;
23   }
24 }
```

Listing 2: Testing Class Example

```
1  public class CodeTest {
2
3    @Test public void test0() {
4      CodeClass base = new CodeClass();
5      assertNotEquals(0, base.getDivisor());
6    }
7
8    @Test public void test1() {
9      CodeClass base = new CodeClass();
10     assertEquals(0, base.div1());
11   }
12
13   @Test public void test2() {
14     CodeClass base = new CodeClass();
15     assertEquals(0, base.div2());
16   }
17 }
```

Listing 1 presents a Java class, `CodeClass`, that computes division results, and Listing 2 presents its JUnit test class, `CodeTest` that tests `CodeClass`, with three test methods.

`CodeClass` includes two getters: `getDividend` and `getDivisor`, used to perform division. Meanwhile, `div1` and `div2` are different implementations of division: `div1` is equipped with a precondition check that examines if the divisor is zero while `div2` lacks any such precondition.

In `CodeTest`, `test0` performs an explicit test to ensure that the divisor (`getDivisor`) is never zero. `test1` invokes `div1` to verify if the resulting division is as expected. Finally, `test2` executes `div2` to perform a test similar to `test1`.

In this test suite, all tests pass for `CodeClass`. No oracles are in violation: neither the preconditions in the source code nor the

assert statements in the tests. However, if we generate a mutation that mutates the return value of *getDivisor* from 3 to 0, the mutant gets killed, but the cause of its death could be interesting:

For such a mutation at Line 4 in `CodeClass`, all three test cases fail, but for different reasons.

(1) `test0` fails due to a test oracle that requires the divisor to be non-zero.
(2) `test1` fails due to a source code oracle: an intentional pre-condition check that throws as an *AssertionError* in `div1`, on Line 15 in the source code.
(3) Finally, `test2` fails due to a system-triggered crash, specifically, an uncaught runtime exception: the Java Runtime Environment (JRE) throws an *ArithmeticException* at Line 22 in `div2`, leading to the test failure.

The code example, while contrived and trivial, illustrates a potential characteristic of mutation testing: a single mutant may die by different tests, for different causes. In Section 2.3, we highlight prior works that confronted some issues concerning the causes of test failures in relation to mutation kills.

In this paper, we consciously do not make claims as to which types of test failures are "better" or "worse." In fact, we can imagine reasons to attempt to target each type, depending on the application context for mutation testing. But, absent an understanding and study of the prevalence of these forms of test failure, each application could be affected in unforeseen ways.

Indeed, we find that while mutation testing was originally developed to strengthen a test suite (and its test oracles), it has found other useful applications. For instance, when applying mutation testing to replicate crashes (or uncaught exceptions) occurring in the wild, it may become useful to track mutants that trigger crashes or uncaught exceptions. Similarly, when practitioners employ defensive programming practices like using source-code oracles (*e.g.,* in-code pre-/post-conditions, or boundary value checks) it becomes useful to know if mutants are killed due to source code oracles.

2.3 Prior Works on Ease of Mutant Kills

Mutation testing is used for multiple applications such as test suite evaluation and improvements, generation of test data and test suites, and crash replication. The quality of mutants — assessed by the difficulty in killing a mutant — is vital to applications of mutation testing. In particular, a mutant that is hard to kill reveals gaps in test data, test oracles, and program logic that were previously unknown or not obvious. As such, a hard-to-kill, or "stubborn" mutant necessitates additional test oracles with more test data, or whole new tests to kill the mutant. Stubborn mutants may also reveal exceptional program crashes, or boundary value faults that are otherwise hard to reproduce with existing tests and data.

Stubborn Mutants. Prior works prefer "stubborn" or hard-to-kill mutants, especially for test generation [7], evaluating test suites [31], or prioritizing mutant selection [17, 39, 42]. Such works tend to apply thresholds on mutant killing rates to classify mutants as stubborn. A mutant is considered stubborn if it can be killed by at most a certain threshold of tests in a test suite. Different works applied different thresholds on killing test counts to classify stubborn mutants [21, 27, 39, 40].

Trivial Mutants. In contrast, easily killed mutants, especially mutants killed by numerous covering tests, are a lesser focus in mutation testing research. When modeling "mutant utility," Just *et al.* [20] define trivial mutants as those that cause all covering tests to crash due to an (uncaught) exception. Indeed, multiple definitions for trivial mutants have emerged in prior works [22, 24, 38].

Even while identifying stubborn and trivial mutants, the focus of such prior works was not to perform a systematic study of stubborn or trivial mutants, or how such mutants died. Such works did not consider the causes of test failures that lead to a mutant kill (*e.g.,* test oracles, source-code oracles, crashes). In the same vein, in our literature review we find no prior study addressing the notion that the same mutant can be killed in different ways (crash vs. oracles) by multiple tests; and the impact that such a notion can have on the applications of mutation testing.

We posit that mutant triviality or stubbornness is tied to the stylistic choices that developers make when writing code. As such, capturing mutant triviality or stubbornness with only counts of failing tests may be inadequate. Studying the underlying causes of the test failures in a mutant's kills is vital.

For instance, one cannot easily conclude that a heavily (or easily) killed mutant is uninteresting without understanding how they are killed, or without understanding that they can be possibly killed due to various causes. Just because all tests that execute a mutant fail due to thrown exceptions does not mean that the mutant is trivial. Often, production code involves developer-informed oracles in source code. As shown in Listing 1, such a "source-code oracle" can take the form of either an *Assert*-based pre-/post-condition check, or an explicit thrown exception instantiated in the production code. The resulting exception-induced kills may have varying causes since they may be triggered by different source-code oracles, each effectively functioning as an in-code test case.

3 APPROACH: DETECTING AND CLASSIFYING CAUSES OF TEST FAILURES

In this section, we define how a test fails due to (a) source code oracle, (b) test oracle, and (c) crash, and we describe the approach that we use to detect failures by those three causes of test failure.

3.1 How Are Exceptions Instantiated, Thrown, and How Do They Cause Test Failures?

In Java projects with test cases using the JUnit framework, a test fails with at least one uncaught *Throwable* instance[2]. The occurrence of an exception or an error, as represented by the *Throwable* class, can result in test failures. Such error and exception instances are freshly created in the context of the exceptional situation and are instantiated with context information stored in stack trace [28]. A *Throwable* instance stores a snapshot of the execution stack of its thread at the time it was created [28].

Therefore, to categorize types of test failures, it is necessary to determine where and how an uncaught throwable object was instantiated and thrown. A *Throwable* instance could be instantiated from some code, located in some code, and be thrown by some code.

[2]Some methods with specific annotations, such as `@After`, are guaranteed to run even if a `@Before` or `@Test` method throws an exception.

Based on our classification, a *Throwable* object could be thrown from:

(1) project code with *throw* statements, including project production code and project test code;
(2) third-party packages with *throw* statements that come with Java platform; or
(3) the Java Virtual Machine.

Stack traces provide the context of the exceptional situation that helps locates the fault. An exceptional context could be located in:

(1) project production code (project source code) and project test code;
(2) other Java packages; or
(3) nowhere: no exceptional context is available due to special reasons, such as running out of memory.

A *Throwable* object could be instantiated through:

(1) current project's explicit instantiation of a *Throwable* instance;
(2) another package's instantiation of a *Throwable* instance; or
(3) the Java Virtual Machine

In the next two subsections, we describe how we determine such information.

3.2 Subject Instrumentation

To determine the necessary information listed above to classify test failures, we instrument the bytecode of the software under test. Although we expect that projects define some of their own exceptional classes, the fact is: many projects throw exceptional classes that were defined in JCL (Java Class Libraries). Therefore, classifying a test failure as a "source-code oracle," based only on whether the exception type was defined in first-party code is not ideal and would miss many such in-code checks. Instead, we use ASM bytecode-manipulation library [6] to trace the instantiation of exceptional classes. We place instrumentation probes after the instantiation of all exception classes. As an initial heuristic, we recognize these in the bytecode as instantiations of classes whose names end in "Exception" or "Error", which we will address below. The probes record the exceptional context, including file name and line number, which are going to be matched with the top element of the failure stack trace of the failing test. Such matches indicate source-code oracle failures.

We also place probes that record exceptional information after the instantiation of exceptional classes in the project's test code. Some test cases have sanity checks by using a direct *throw* in test cases to assert the absence of an exception without using assertions provided by testing APIs. Additionally, try-catch blocks are prevalent in test cases. Ma *et al.* [23] reported that 16.5% JUnit test cases contain try-catch statements. Thus, if an instantiation of an exception in test code directly leads to a test failure, we consider the failure as a test-oracle failure.

Admittedly, the above method can produce both false positives and false negatives when recognizing exceptional classes based on class name, and we properly account for both situations. For false positives, non-exceptional classes ending with "Error" and "Exception" that do not lead to test failures may be instrumented. However, because they do not cause any failures, the instrumentation traces are simply not matched or used and thus will have no impact on our failure-cause analysis. To address false negatives stemming from classes that do not end with "Error" or "Exception" that are inherited from Throwable and do lead to test failures, we recognize these, post hoc, by analyzing the types of uncaught exceptions that PIT reports as the cause of failure.

Finally, if a test failure is not recognized as test-code oracle failure or source-code oracle failure, it is labeled as an exogenous crash.

3.3 Modified PIT for Mutation Testing

We use PIT [8], a state-of-art open-source bytecode mutation testing tool for Java, in our experiments. PIT runs the test suite and collects coverage information without any mutations during an initial phase, and then selects only test cases that potentially cover the mutations to speed up the execution time. In this experiment, we used all of the normal mutation operators provided by PIT.

To acquire test failing reasons from PIT, we modified the *onTestFailure* method in *CheckTestHasFailedResultListener.java* in PIT version 1.9.5. We print each uncaught Throwable instance's exception type and the top element of the stack trace (if available) to Standard Error, which includes the class name, file name, method name, and line number.

PIT firstly collects basic block code coverage to determine which test to run and labels mutants with no covering tests as *NO_COVERAGE* without running any test cases. Then it labels all covered mutants as SURVIVED or KILLED while labeling a proportion of mutants as *MEMORY_ERROR* or *TIMED_OUT* due to technical and performance issues. For such reasons, the status of those relevant test runs is not included in our results.

3.4 Classifying Causes of Test Failures

We provide the following rules for classifying a failing test based on failing reasons in the following order.

(1) If an uncaught *Throwable* Instance is of the project domain's type, the test case fails due to a **source-code oracle**.
(2) If an uncaught *Throwable* Instance is of a type from the testing library, including JUnit, Mockito, and so on (by examining the type of exceptions and the files that throw them), the test case fails due to a **test failure**.
(3) If an uncaught *Throwable* instance is not of any type above, then
    - If this instance's exception type and the top element's stack information (line number, and file name) match the information specified in our probe for project source code, we consider the test case to have failed due to a **source-code oracle**.
    - If this instance's exception type and the top element's stack information (line number, and file name) match the information specified in our probe for project test code, we consider the test case to have failed due to a **test failure**.
(4) If a failing test does not apply to any situations above, we consider the test to have failed due to an **exogenous crash**.

4 EXPERIMENTAL SETUP

In this section, we outline the experimental design to answer key research questions about how mutants are executed and killed by

test cases. We start by enumerating four research questions, along with the rationale for why we ask those questions. Next, to answer those questions we performed mutation testing for 10 open-source Java programs — we provide the specifics about those programs and their version numbers to help with replicating this experiment. We carried out the mutation test runs using the modifications we made to PIT, as discussed in Section 3. During those test runs, we tracked test-failure data to compute experimental variables used to answer the research questions below.

4.1 Research Questions

In an effort to better understand mutation testing kills and to give direction and focus to our investigation, we devised the following set of research questions:

RQ1 How many tests execute a given mutation and what are their failure rates?
RQ2 How does a test kill a mutant?
RQ3 Are certain mutation operators more prone to certain test failure causes?
RQ4 What types of uncaught exceptions kill mutants?

These research questions are aimed at understanding how mutant kills occur. The questions start with examining how multiple tests execute a mutant. We then progress to examine the causes of mutant kills and end with an exploration of factors behind the different causes of mutation kills. We now elaborate on the rationale for these questions, starting with *RQ1*.

RQ1*: How many tests execute a given mutation and what are their failure rates?* Past research has given labels to and assigned relative merit to mutations that cause various failure rates. Section 2.3 discusses the prior research that labeled mutants as "stubborn" if they were executed by multiple test cases and failed by a threshold-limited percentage of test cases, or research that labeled mutants as "trivial" if they were executed by multiple test cases and failed by a too great percentage of test cases. To empirically understand how often these conditions actually occur and thus inform future mutation-testing research and uses thereof, we examine this dichotomy in mutation testing by examining two phenomena: (a) multiple tests executing a single mutant; and (b) the failure rates of tests, per mutant; with the variables enumerated below.

Experimental Variables:

- Number of Multi-Test Mutants. This variable will track the occurrence of multi-test mutants, *i.e.,* mutants that are executed by multiple tests, in a subject program's test suites. In addition to tracking the number of Multi-Test Mutants for a subject, we will break down those counts by:
  - Survived: Number of multi-test mutants that were not killed by any executing tests.
  - Wholly Killed: Number of multi-test mutants that were killed by *all* of the executing tests.
  - Partially Killed: Number of multi-test mutants that were killed by a part of the executing tests.

RQ2*: How does a test kill a mutant?* Mutation testing's original motivation was to assess the strength of tests in a test suite [1, 9, 15]. Such a motivation would require that tests, when killing mutants, fail due to test oracles. In *RQ2*, we examine if mutants necessarily are killed due to failing test oracles. Specifically, we investigate the degree to which the following causes of test failures contribute to mutation kills: (a) test oracles; (b) source-code oracles; and (c) exogenous crashes. Indeed, as we discuss in Section 2, different causes of test failures that lead to mutant kills, would invite different motivations for performing mutation testing.

Experimental Variables: For each type of failure cause (*i.e.,* (a) test oracles; (b) source-code oracles; and (c) exogenous crashes), we track the following metrics:

(1) Number of Test Cases for each Failure Cause. This variable counts the number of test cases that fail due to each of the three failure causes, computed for each subject program. Notably, if a test case encounters multiple causes of failure across different test runs (for different mutations), we count the test once for each unique cause of failure.
(2) Number of Test Runs for each Failure Cause. This variable counts the number of test executions that fail due to each of the three enumerated failure causes. A single static test case can be executed for different mutants, to produce multiple test runs. Further, note that each single test run can fail due to one failure cause.

RQ3*: Are certain mutation operators more prone to certain failure causes?* After studying how tests execute and kill mutants, we examine any relationship between the causes of mutant kills and the mutants themselves. Specifically, we examine how different mutation operators induce test failures of various types (with differing causes). For instance, results of *RQ3* should help us assess if certain mutation operators are more likely to induce exogenous crashes than others. If so, applications of mutation testing can use such results to include (or exclude) such mutation operators from mutation testing runs for specific programs.

RQ4*: What types of uncaught exceptions kill mutants?* Our data suggest that mutants are often killed by uncaught runtime exceptions. This leads us to examine the types of runtime exceptions that trigger test failures, and thus, mutant kills. Note that runtime exceptions may be triggered by source-code oracles or by exogenous factors (*e.g.,* the Java Runtime Environment).

4.2 Subject Programs

We ran our experiments on 10 open-source Java projects. We selected well-known, real-world projects that are built with Maven, have unit tests written using the JUnit test framework by developers, and whose source code is available on GitHub. All studied programs use the JUnit testing framework and have JUnit on the classpath. Each column of Table 1 provides: (1) subject project, (2) version analyzed, (3) lines of code, (4) number of tests involved (excluding tests that failed on the unmutated program), (5) line coverage (*LC*), and (6) mutation score (*MS*).

When selecting subject programs we excluded projects with substantial numbers of failing tests (before any mutation). We also excluded projects that included other factors that would be not suitable for mutation analysis, such as implicit test-order dependencies or shared file access on disk. After the selection process, we also configured PIT to exclude any failing tests when running without any mutations from our experiments. Additionally, we configured

Table 1: Experiment Subject Programs

Subject Program Name	Version	LoC	#Tests	LC	MS
commons-lang	3.12.0	84,788	4,967	95%	85%
commons-jexl	3.2.1	31,321	790	76%	59%
joda-time	2.12.2	88,190	5,501	90%	80%
commons-text	1.10.0	26,579	1,242	99%	84%
commons-io	2.11.0	40,379	1,771	89%	79%
xmlgraphics	2.7	35,355	188	42%	28%
gson	2.9.1	26,685	1180	89%	80%
jsoup	1.15.2	25,985	900	83%	62%
commons-csv	1.9.0	8,007	390	98%	93%
commons-validator	1.7	16,781	533	78%	73%

the JVM to enable assertions in the source code and to enable full stack trace information when a test fails. Finally, we configured PIT to not generate mutations on our instrumented code.

5 RESULTS

RQ1: Failure Rates for Multi-Test Mutants

The results for *RQ1* are presented by Table 2. For each subject program, the first three columns report: (a) All Mutants: the total number of mutants whose data we tracked as part of our experiments; (b) Single-Test Mutants: a subset of All Mutants that were executed by only one test; and (c) Multi-Test Mutants: a subset of All Mutants that were executed by at least two or more tests. In addition, Table 2 reports the number of Multi-Test Mutants as a percentage of All Mutants. For instance, Table 2 shows that in GSON we tracked 2549 mutants and found that 2303 (or 90.35%) of the 2549 mutants were executed by two or more tests.

Further, Table 2 breaks down the counts of such Multi-Test Mutants into three categories: Survived; Wholly Killed; and Partially Killed. As such, once again for GSON, we see that out of the 2303 Multi-Test Mutants: 235 Survived; 678 were Wholly Killed by causing all test cases that executed those mutants to fail; and 1390 were Partially Killed, *i.e.,* not all tests executing those mutants failed. Further, the Survived mutants account for 9.22% of all 2303 mutants we tracked in GSON; Wholly Killed mutants account for 26.6% of All Mutants; and Partially Killed mutants amount to 54.53% of all mutants in GSON.

We find that the number of Multi-Test Mutants form the majority of mutants, across all ten subject programs. The proportion of Multi-Test Mutants ranges from 90.35% in GSON to 62.36% in Commons-Lang. Further, in nearly all programs, the majority of such multi-test mutants are killed "partially" by some of the mutants' executing tests (except Commons IO and Commons-Lang). Finally, we also find that a smaller proportion of multi-test mutants do *survive*, *i.e.,* all tests executing such mutants pass.

RQ2: How Does a Test Kill a Mutant?

The results for *RQ1* suggest that substantial proportions of the mutants in our subject programs are typically executed by multiple tests. And as discussed in Section 2.3, multiple tests executing the same mutant may result in failures by different causes.

As such, using Figure 3 and Table 3, we summarize the results for *RQ2*. Table 3 breaks down the *failing tests* into the three failure causes: (a) test oracles; (b) source-code oracles; and (c) exogenous crashes. Whereas, Figure 3 summarizes the *killed mutants* across all subject programs with their failure causes.

For each subject, Table 3 shows the Number of Test Cases impacted by different failure causes. For instance, Commons-Lang has 4452 tests failing on mutations due to test oracles; 1487 tests failing on source-code oracles and 3527 tests failing due to exogenous crashes. Further, for each count of test cases, Table 3 also shows the number of Test Runs for each failure cause, *i.e.,* test executions that failed when executing a specific mutation. For instance, in Commons-Lang, the 4452 tests that fail due to test oracles, fail across 60,897 test runs. Specifically, those 4452 tests were executed repeatedly for different mutations and collectively failed 60,897 times due to failing test oracles. Similarly, the 1487 tests that are failing due to source oracles, result in 11,101 failing test runs; while 3527 tests cause 26,177 crashing test runs.

Additionally, Table 3 shows the relative percentages of the Number of Failing Test Runs. Notably, these results show the outsized role of exogenous crashes in test runs that execute and fail on mutations. Exogenous crashes can account for as many as 46.22% and 37.33% of failing test runs in Jsoup and Commons-Jexl, respectively. And among the subjects in our experiments, they contribute to at least 22.11% of failing test runs, as in the case of Commons-Text. Further, test oracles do not always form the majority failure cause for a set of failing test runs. Commons-Jexl shows that test oracles can cause as little as 11.82% of test runs to fail in a program's mutation test runs.

Additionally, Figure 3 illustrates the Failing Test Runs data by attributing individual mutants to test failure causes. Specifically, Figure 3 uses a grid of Venn diagrams — one per subject program — to show how mutations introduced in a program are killed by any of the three failure causes: test oracles, source-code oracles, or crashes; and often are killed by multiple causes (*i.e.,* multiple failing test cases for a given mutant, and those failing tests differ in their failure cause).

Again, consider the Venn-diagram for Commons-Jexl in Figure 3 (second in the first row). The Venn diagram for Commons-Jexl clearly shows that 1146 mutants, *i.e.,* 21.3%, of all killed mutants we tracked getting executed by test runs that failed because of all three failure reasons: test oracles, source-code oracles, or crashes. Notably, every subject program has mutants that were killed by all three test failure causes. Additionally, across all subjects, the number of mutant kills due to exogenous crashes (shown as red bubbles) forms a substantial portion of the mutants for a test subject. Again, for Commons-Jexl, 2281 mutants (42.3% of all killed mutants in Commons-Jexl) can be killed due to crashing test runs; of which 1146 can be killed by all three failure causes; 247 mutants can be killed either by crashes or source-code oracles; 340 mutants can be killed by test-oracles and crashes; while 548 mutants can be killed by crashes alone.

RQ3: Are Certain Mutation Operators More Prone to Certain Failure Causes?

To understand the influence of mutation operators on killing states, we separate implicit kills, *i.e.,* test failures due to exogenous crashes, from explicit kills, *i.e.,* test-oracle failures or source-code oracle failures. We present the percentage of exogenous crashes among

Table 2: Mutants covered by multiple tests. (Percentages in the table use "All Mutants" counts as the denominator.)

			Mutants Executed by Multiple Tests			
Program Name	All Mutants	Single-Test Mutants	Multi-Test Mutants	Survived	Wholly Killed	Partially Killed
gson	2,549	246 (09.65%)	2,303 (90.35%)	235 (09.22%)	678 (26.60%)	1,390 (54.53%)
commons-jexl	7,085	744 (10.50%)	6,341 (89.50%)	1,466 (20.69%)	1,479 (20.88%)	3,396 (47.93%)
commons-csv	654	72 (11.01%)	582 (88.99%)	27 (04.13%)	211 (32.26%)	344 (52.60%)
jsoup	4,278	528 (12.34%)	3,750 (87.66%)	855 (19.99%)	965 (22.56%)	1,930 (45.11%)
joda-time	8,859	1,793 (20.24%)	7,066 (79.76%)	849 (09.58%)	2,274 (25.67%)	3,943 (44.51%)
commons-io	3,768	787 (20.89%)	2,981 (79.11%)	425 (11.28%)	1,436 (38.11%)	1,120 (29.72%)
commons-validator	1,760	366 (20.80%)	1,394 (79.20%)	185 (10.51%)	473 (26.88%)	736 (41.82%)
commons-text	4,240	1,062 (25.05%)	3,178 (74.95%)	389 (09.17%)	1,381 (32.57%)	1,408 (33.21%)
xmlgraphics	3,854	1,336 (34.67%)	2,518 (65.33%)	577 (14.97%)	932 (24.18%)	1,009 (26.18%)
commons-lang	12,441	4,683 (37.64%)	7,758 (62.36%)	838 (06.74%)	3,781 (30.39%)	3,139 (25.23%)

Table 3: Number of test runs and the underlying test cases that can fail due to different failure causes. (Percentages in the table use "# of All Failed Test Runs" as the denominator.)

		Counts of Tests (and Test runs) by Failure Causes					
	# of All Failed	Due to **Test Oracles**		Due to **Source Oracles**		Due to **Crashes**	
Program Name	Test Runs	# of Tests	# of Test Runs	# of Tests	# of Test Runs	# of Tests	# of Test Runs
commons-lang	98,175	4,452	60,897 (62.03%)	1,487	11,101 (11.31%)	3,527	26,177 (26.66%)
commons-jexl	390,973	770	46,209 (11.82%)	774	198,806 (50.85%)	786	145,958 (37.33%)
joda-time	349,514	4,958	140,368 (40.16%)	4,157	83,365 (23.85%)	4,554	125,781 (35.99%)
commons-text	45,315	1,191	23,874 (52.68%)	669	11,421 (25.20%)	976	10,020 (22.11%)
commons-io	33,586	1,549	16,340 (48.65%)	996	7,772 (23.14%)	1,330	9,474 (28.21%)
xmlgraphics	8,562	173	3,306 (38.61%)	99	2,621 (30.61%)	156	2,635 (30.78%)
gson	117,845	1,104	26,072 (22.12%)	1,129	50,434 (42.80%)	1,135	41,339 (35.08%)
jsoup	223,145	877	77,517 (34.74%)	855	42,483 (19.04%)	858	103,145 (46.22%)
commons-csv	16,092	362	7,842 (48.73%)	244	2,831 (17.59%)	339	5,419 (33.68%)
commons-validator	20,952	485	12,977 (61.94%)	169	1,837 (08.77%)	447	6,138 (29.30%)

Figure 3: Relative number of killing reasons for mutants, including mutants killed for by combinations of failure types.

all failing tests for different projects in Table 4. We exclude the *InvertNegs* mutator in this table because few mutations are generated by this mutator thus few failing tests are involved.

We find that 48.81%–93.18% of failing tests that cover mutants from the *Increments* mutator are exogenous crashes. 48.16%–84.0% test runs regarding *NullReturnVals* result in exogenous crashes among all projects. Overall, we find that the *Math* mutator, *ConditionalsBoundary* mutator, *NullReturnVals* mutator, *Increments* mutator, *NegateConditionals* mutator usually result in much more exogenous crashes. In turn, the *PrimitveReturns* mutator, *BooleanFalseReturnVals* mutator, *VoidMethodCall* mutator, *BooleanTrueReturnVals* mutator, and *EmptyObjectReturnVals* mutator are less prone to violate checks in the Java virtual environment. However, such trends are not always true for all projects. For example, the *VoidMethodCall*

Table 4: Percentages of *3rd Party Crashes* among all failing tests for different mutation operators.

Mutator	commons-lang	commons-jexl	joda-time	commons-text	commons-io	xml-graphics	gson	jsoup	commons-csv	commons-validator
PrimitiveReturns	02.82	05.81	07.40	03.46	01.36	14.93	00.72	15.24	00.48	00.00
Math	21.82	51.36	22.20	46.51	42.99	19.11	39.37	75.96	70.09	43.78
ConditionalsBoundary	56.77	61.47	49.85	22.39	22.98	53.38	82.80	63.56	89.40	48.96
NullReturnVals	68.89	70.16	78.78	76.85	78.37	57.82	61.38	78.59	84.00	48.16
BooleanFalseReturnVals	05.22	02.51	14.92	00.15	00.34	14.67	23.78	31.87	01.02	00.00
VoidMethodCall	08.97	50.26	09.91	05.51	19.56	16.03	02.16	36.18	04.66	21.60
BooleanTrueReturnVals	03.39	01.19	22.63	04.55	16.05	14.72	27.97	47.56	08.48	08.67
NegateConditionals	21.66	33.82	25.59	10.70	19.06	28.47	33.67	39.93	21.68	27.62
EmptyObjectReturnVals	07.15	35.80	02.80	00.95	11.45	11.99	10.53	23.36	38.25	29.25
Increments	70.53	93.18	57.99	49.90	62.04	48.81	85.40	49.23	85.71	50.00

mutator, which replaces a method call with a void method, generally generates fewer exogenous crashes. However, this mutator leads to 50.26% crashes and 36.18% crashes among failing tests in *commons-jexl* and *jsoup*, respectively.

RQ4: What Types of Uncaught Exceptions Kill Mutants?

In Table 5, we list the top-5 exception-types from exogenous crashes for each project. We observe that *NullPointerException* ranks as the top crash-/exception-type across all projects. Prior work [26, 36] has identified similar results about *NullPointerException*-induced failures. Violation of the index boundaries, including *ArrayIndexOutofBounds*, *StringIndexOutofBounds*, and *IndexOutofBounds* is another major contribution of exogenous crashes. Among them, *ArrayIndexOutOfBoundsException* ranks 2nd in 5 projects and is in the top-5 categories for all projects in this study.

While referencing null values and index-boundary checks are two major causes for the Java virtual machine to throw an exception in mutation testing, we also show the proportion of the top-3 mutator-exception pairs among all exogenous crashes from Java class libraries in Figure 4.

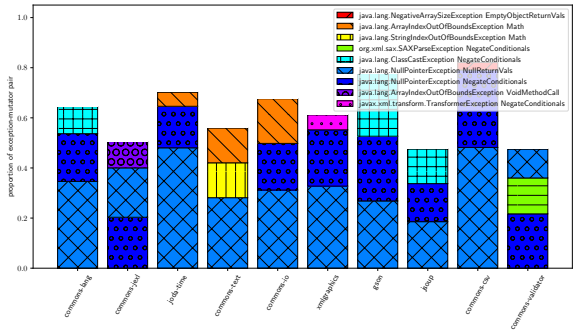

Figure 4: Top 3 mutator-exception pairs for *3rd Party Crashes*.

In Figure 4, we find that some specific mutator-exception pairs account for a sizable proportion of exogenous crashes. Test cases usually fail with *NullPointerException* when running tests for *NegateConditionals* mutator and *NullReturnVals* mutator, which contributes to 50% (approx.) of all exogenous crashes led by these two mutators. One intuition is evident: the *NullReturnVals* mutator mutates the return value to *null*, and if the tested program were written carefully, it would have checked for *null* before dereferencing the returned value. Such defensive programming would have moved the failure type from an uncontrolled implicit failure (*i.e.,* exogenous crash due to a null dereference) to a controlled explicit failure (*e.g.,* a source-code oracle, which explicitly throws an exception), or perhaps even could have avoided failure altogether by properly recovering from the erroneous state.

6 DISCUSSION

Through our experiments we have uncovered several key findings that challenge assumptions about mutation testing. We discuss our findings below, followed by an enumeration of their implications.

First, not all uncaught exceptions are due to implicit checks. Exogenous crashes are prevalent in mutation testing, though assertion violations are mainly expected by practitioners when applying mutation analysis. Also, one should not assume that all explicit checks are oracles in the test code.

Second, oracles in source code are designed intentionally by developers with error messages, no matter if it is in the form of an `Assert` or an exception being thrown. Furthermore, when we separate source-code oracle violations from all exogenous crashes, we find such checks in source code are non-negligible in contributing to the mutation score. Therefore, the mutation score indicates the fault detection capabilities for explicit checks in the source code.

Third, certain mutation operators are prone to lead to exogenous crashes. For example, many killing test runs for the *NullReturnVals* mutator fail due to NullPointerExceptions. We think such trends can give advice in selecting mutation operators when we insert faults into programs for different purposes. Also, it alerts us that we might increase the complexity of mutation operators, when possible. For instance, instead of simply mutating the return value to null, empty object, or zero, one might consider mutating the return value to other non-zero values or objects with customized field values for specific projects. Such practices might lower the ratio of exogenous crashes, thus improving the quality of the mutation score.

Practical Implications. Our results and findings bear potentially significant implications for both researchers and practitioners of mutation testing. We organize our practical implications into the following eight themes.

(1) **Researchers should take a fresh look at mutation testing, particularly for the original purpose of assessing test-suite effectiveness.** A sizable portion of the kills happen due to crashes (*i.e.,* not test oracle failures), and this fact

Table 5: Top 5 categories of *3rd Party Crashes*

Project	Top-5 categories	Project	Top-5 categories
commons-lang	java.lang.NullPointerException (57.8%) java.lang.StringIndexOutOfBoundsException (14.0%) java.lang.ClassCastException (10.8%) java.lang.ArrayIndexOutOfBoundsException (10.3%) java.lang.UnsupportedOperationException (1.1%)	commons-jexl	java.lang.NullPointerException (42.0%) java.lang.ArrayIndexOutOfBoundsException (31.2%) java.lang.StackOverflowError (6.7%) java.lang.StringIndexOutOfBoundsException (4.7%) java.util.concurrent.ExecutionException (3.2%)
joda-time	java.lang.NullPointerException (67.2%) java.lang.ArrayIndexOutOfBoundsException (13.2%) java.lang.NoClassDefFoundError (10.5%) java.lang.StringIndexOutOfBoundsException (2.8%) java.lang.StackOverflowError (1.9%)	commons-text	java.lang.NullPointerException (36.4%) java.lang.ArrayIndexOutOfBoundsException (29.0%) java.lang.StringIndexOutOfBoundsException (18.2%) java.lang.IllegalArgumentException (7.2%) java.lang.IndexOutOfBoundsException (3.5%)
commons-io	java.lang.NullPointerException (52.8%) java.lang.ArrayIndexOutOfBoundsException (24.2%) java.nio.file.DirectoryNotEmptyException (8.1%) java.lang.IndexOutOfBoundsException (2.8%) java.lang.StringIndexOutOfBoundsException (2.2%)	xmlgraphics	java.lang.NullPointerException (61.8%) java.lang.ArrayIndexOutOfBoundsException (8.3%) javax.xml.transform.TransformerException (7.8%) java.lang.ClassCastException (3.5%) java.lang.StringIndexOutOfBoundsException (2.7%)
gson	java.lang.NullPointerException (55.8%) java.lang.ClassCastException (26.3%) java.lang.ArrayIndexOutOfBoundsException (9.4%) java.lang.IndexOutOfBoundsException (3.1%) java.lang.StringIndexOutOfBoundsException (2.1%)	jsoup	java.lang.NullPointerException (41.3%) java.lang.ClassCastException (17.5%) java.lang.IndexOutOfBoundsException (12.4%) java.lang.StackOverflowError (11.7%) java.lang.ArrayIndexOutOfBoundsException (9.6%)
commons-csv	java.lang.NullPointerException (81.4%) java.lang.StringIndexOutOfBoundsException (6.6%) java.lang.ArrayIndexOutOfBoundsException (6.3%) java.lang.NegativeArraySizeException (3.1%) java.lang.UnsupportedOperationException (1.2%)	commons-validator	java.lang.NullPointerException (34.0%) org.xml.sax.SAXParseException (32.1%) java.lang.ArrayIndexOutOfBoundsException (11.6%) java.util.NoSuchElementException (8.5%) java.lang.StringIndexOutOfBoundsException (4.0%)

challenges a prime motivation of mutation: improving the quality of tests.

(2) **It sheds light on a hidden way of killing mutants for both researchers and practitioners.** Rather than augmenting existing test suites, practitioners could consider adding useful oracles in source code, which could be checked by multiple tests.

(3) **The study helps both practitioners and researchers understand what mutation score is actually composed of and understand the implications of traditional mutation score.** Mutation testing assesses more than the fault detectability of the test suite itself; oracles in source code can sometimes contribute substantially to mutation scores.

(4) **It sheds light on new possible research techniques.** While several research techniques focus on killing surviving but killable mutants by automatically augmenting existing test suites by adding test assertions, few suggest adding source-code oracles to kill a mutant. Approaches like dynamic invariant detection [4, 11, 12] may be deployed to identify invariants that could be checked by such source-code oracles during mutation runs, consequently leading to the additional kills of surviving mutants.

(5) **It calls for more research studies to investigate the difference and relative importance of the three distinct failing reasons in mutation analysis.** The relative importance, in terms of assessing test-suite strength, bug-detection sensitivity, and so on, of the different ways that mutants are killed remains to be investigated.

(6) **This study suggests the possibility of a weighted mutation score for evaluating different aspects of a test suite.** Test assertions are traditionally assumed as the source of mutation kills, however, these are the final check during execution after several runtime checks and source-code oracle checks. Thus, a new mutation metric may provide variable weighting for each type of kill, and the effects of such could be investigated.

(7) **The definition of ease of mutant kills calls for reconsideration.** Prior work has applied thresholds on mutant killing rates through heuristics to determine if mutants are stubborn or trivial [21, 22, 24, 27, 38–40]. However, our findings suggest that a variety of causes for mutant failures could potentially explain a portion of the perceived triviality and stubbornness of mutants.

(8) **Our reported mutator-crash-pair results may support different objectives of using mutation.** Techniques that augment the existing test suite by adding new test oracles can possibly focus on choosing mutation operators that lead to fewer exogenous crashes. In contrast, objectives like crash replication might favor crash-inducing mutation operators.

7 THREATS TO VALIDITY

Threats to internal validity may arise from the empirical setup that we designed. We classified test-failure causes based on the types of exceptions associated with the failures, the location of exception instantiation, and the stack trace information. It is possible that the stack-trace information could possibly be intentionally modified for various reasons, thus making our results less accurate.

The primary external threats to our experimental validity stem from the generalizability of our results. Our mutation testing experiments focused on the unit tests for ten Java programs that used the JUnit testing framework. Although other programming languages may be tested differently and may use other mutation operators, our approach could easily be extended to those languages and testing frameworks. Indeed, our experiment is limited to ten programs, as such we are unable to definitively extrapolate our findings to other programs. Additionally, there might exist some flaky tests in those programs, leading to flaky results in our experiment [37]. However, our experiment's subject programs are real-world, open-source

projects with mature test suites. As such, we are confident that these results provide substantive evidence that multiple test-failure causes do contribute to mutation testing results, and thereby warrant serious consideration. Future work will extend such studies to other languages and platforms to determine the extent of the differences. Moreover, we plan on replicating our study with other mutation testing frameworks, as part of future work.

Threats to construct validity in our study could originate from the mutation testing tool employed. We utilized a single mutation testing tool, PIT, for generating mutations in our experiments, rendering our study susceptible to implementation choices and faults in PIT. However, PIT is a mature mutation testing framework that is commonly used across practice and research [16, 37, 43].

8 RELATED WORK

Schuler *et al.* [36] devised a new metric called *checked coverage* to evaluate oracle quality, which considers the backward dynamic slice of covered statements that actually influence an oracle. They found that a test suite with no assertions still detects over 50% of the mutations detected by the original test suite. Around 80% of these detected mutations are caught by implicit checks (uncaught exceptions). They also found that 32–65% of the mutations are detected by implicit checks, which is mainly due to NullPointerExceptions caused by mutations. However, we consider all defensive programming practices, *i.e.,* explicit expected exceptions thrown in first-party source code, not limited to *AssertionError*.

Schuler *et al.* [35] also evaluates the impact of mutants by checking the reasoned dynamic invariants from variables accessed by methods in test execution. And they also found that mutations that violate invariants are significantly more likely to be detectable by a test suite. In contrast, in this work, we assess actual developer-written oracles rather than potentially inferred oracles.

Linares *et al.* [22] systematically devise a taxonomy of 262 types of Android faults grouped in 14 categories, identified a set of 38 mutation operators and implemented an infrastructure, *MDroid+* to automatically seed mutations in Android apps with 35 of the identified operators. They found some mutation operators in *Major*, *PIT*, and their *MDroid+* tend to generate more trivial mutants that can relate to *NullPointerException*.

Zhang *et al.* [43] found that assertions are strongly correlated to test-suite effectiveness. They measured the explicit mutation score of a test suite by subtracting the fraction of implicitly killed mutants (ran without assertions) from the original mutation score. However, even the same mutant can be killed due to various reasons. Also, a test case can still fail due to a third-party crash even after removing assertions if it fails due to assertion failure before.

Ma *et al.* [23] evaluated the quality of JUnit test cases. They found 61% of all tests contain more than a single assertion, 16.5% contains try-catch statements and 28.5% test cases use the JUnit testing framework but do not contain any JUnit assertions. They claim this is due to using assertions from testing libraries other than JUnit assertions. In our study, we analyze all assertion failures, including JUnit, Mockito, Hamcrest, AssertJ, commons-truth from different libraries, and a direct *throw* in test code, based on the stack trace information and our probes in test code.

9 CONCLUSION

In this work, we presented a taxonomy for understanding the causes of test failures that contribute to mutation kills in mutation testing. We also created a method to take execution information from actual mutation testing runs and categorize test failures and mutation kills using our taxonomy. We described our open-sourced implementation[1] that enables our empirical study, as well as our experimental data, which is also available for replication.

We found a number of startling results that should give concern to practitioners' interpretations of mutation scores, as well as researchers' use of mutation testing for other automated techniques, or at least, these results should be taken into consideration in such future work. We found that crashes can cause as much as 46.2% of failed test runs and as much as 43.8% of killed mutants (and thus the mutation score), while source oracle failures can contribute as much as 50.9% failing test runs in one subject program. Moreover, regardless of the software system, such crashes substantially contributed to the number of failures (and kills). Additionally, test oracles (which are the traditionally assumed cause of test failures and mutation kills) contributed as little as 11.8% of test failures for one subject. All 10 subjects showed mutants that were killed due to all failure causes from our taxonomy, with one subject exhibiting as much as 21.3% of all killed mutants failing due to all of the failure causes (*i.e.,* the test cases that failed for that mutant failed due to each of the failure causes). Moreover, we report additional findings on the exception types that cause crashes and failures, and find that NullPointerExceptions are consistently the most prevalent type, which gives some potential directions for practitioners seeking to bolster their programs, and also perhaps that source-code oracles may be an effective way to safeguard against such exceptions.

In the future, we will extend these studies to include other programming languages, platforms, and mutation-testing frameworks, as well as extend them to more experimental subjects.

ACKNOWLEDGMENTS

The second author's opinions expressed in this publication are solely his and do not purport to reflect the opinions or views of his employer, Microsoft.

REFERENCES

[1] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1979. *Mutation Analysis.* Technical Report. Georgia Inst of Tech Atlanta School of Information And Computer Science.

[2] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing theoretical minimal sets of mutants. In *2014 IEEE seventh international conference on software testing, verification and validation.* IEEE, 21–30. https://doi.org/10.1109/ICST.2014.13

[3] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE, 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[4] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. 2006. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine, USA) *(ISSTA '06).* Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/1146238.1146258

[5] Leonardo Bottaci. 2010. Type sensitive application of mutation operators for dynamically typed programs. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops.* IEEE, 126–131. https://doi.org/10.1109/ICSTW.2010.56

[6] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.5769

[7] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2021. Killing stubborn mutants with symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–23. https://doi.org/10.1145/3425497

[8] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452. https://doi.org/10.1145/2931037.2948707

[9] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[10] Hang Du, Vijay Krishna Palepu, and James A. Jones. 2023. spideruci/MutationKills: To Kill a Mutant: An Empirical Study of Mutation Testing Kills (ISSTA Replication Package) (v1.0.0). https://doi.org/10.5281/zenodo.7939536

[11] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123. https://doi.org/10.1109/32.908957

[12] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015 Special issue on Experimental Software and Toolkits.

[13] Antonia Estero-Botaro, Francisco Palomo-Lozano, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez, and Antonio García-Domínguez. 2015. Quality metrics for mutation testing with applications to WS-BPEL compositions. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 536–571. https://doi.org/10.1002/stvr.1528

[14] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*. 147–158. https://doi.org/10.1145/1831708.1831728

[15] R. Geist, A.J. Offutt, and F.C. Harris. 1992. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Trans. Comput.* 41, 5 (1992), 550–558. https://doi.org/10.1109/12.142681

[16] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30. https://doi.org/10.1145/3293882.3330559

[17] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2015. How hard does mutation analysis have to be, anyway?. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 216–227. https://doi.org/10.1109/ISSRE.2015.7381815

[18] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 464–475. https://doi.org/10.1109/ASE.2015.14

[19] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678. https://doi.org/10.1109/TSE.2010.62

[20] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring mutant utility from program context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 284–294. https://doi.org/10.1145/3092703.3092732

[21] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating mutation testing alternatives: A collateral experiment. In *2010 Asia Pacific Software Engineering Conference*. IEEE, 300–309. https://doi.org/10.1109/APSEC.2010.42

[22] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 233–244. https://doi.org/10.1145/3106237.3106275

[23] Dor D Ma'ayan. 2018. The quality of junit tests: an empirical study report. In *2018 IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies (SQUADE)*. IEEE, 33–36. https://doi.org/10.1145/3194095.3194102

[24] Phil McMinn, Chris J Wright, Colton J McCurdy, and Gregory M Kapfhammer. 2017. Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas. *IEEE Transactions on Software Engineering* 45, 5 (2017), 427–463. https://doi.org/10.1109/TSE.2017.2786286

[25] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162. https://doi.org/10.1109/ICST.2014.28

[26] Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. 2018. Mdroid+: A mutation testing framework for android. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 33–36. https://doi.org/10.1145/3183440.3183492

[27] Miloš Ojdanić, Wei Ma, Thomas Laurent, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. 2022. On the use of commit-relevant mutants. *Empirical Software Engineering* 27, 5 (2022), 1–31. https://doi.org/10.1007/s10664-022-10138-1

[28] Oracle. 2020. Throwable (Java Platform SE 7). Retrieved April 15, 2023 from https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html.

[29] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628. https://doi.org/10.1002/stvr.1509

[30] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 537–548. https://doi.org/10.1145/3180155.3180183

[31] Matthew Patrick, Manuel Oriol, and John A Clark. 2012. MESSI: Mutant evaluation by static semantic interpretation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 711–719. https://doi.org/10.1109/ICST.2012.161

[32] Goran Petrović and Marko Ivanković. 2018. State of mutation testing at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 163–171. https://doi.org/10.1145/3183519.3183521

[33] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software* 157 (2019), 110388. https://doi.org/10.1016/j.jss.2019.07.100

[34] David S. Rosenblum. 1995. A practical approach to programming with assertions. *IEEE transactions on Software Engineering* 21, 1 (1995), 19–31. https://doi.org/10.1109/32.341844

[35] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient Mutation Testing by Checking Invariant Violations. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/1572272.1572282

[36] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 90–99. https://doi.org/10.1109/ICST.2011.32

[37] 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*. 112–122. https://doi.org/10.1145/3293882.3330568

[38] Ben H Smith and Laurie Williams. 2009. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical software engineering* 14, 3 (2009), 341–369. https://doi.org/10.1007/s10664-008-9083-7

[39] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empirical Software Engineering* 25, 1 (2020), 434–487. https://doi.org/10.1007/s10664-019-09778-7

[40] Willem Visser. 2016. What makes killing a mutant hard. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 39–44. https://doi.org/10.1145/2970276.2970345

[41] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 910–913. https://doi.org/10.1145/2786805.2803206

[42] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th international conference on software engineering*. 919–930. https://doi.org/10.1145/2568225.2568265

[43] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 214–224. https://doi.org/10.1145/2786805.2786858

[44] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. 2018. A systematic literature review of how mutation testing supports quality assurance processes. *Software Testing, Verification and Reliability* 28, 6 (2018), e1675. https://doi.org/10.1002/stvr.1675

Received 2023-02-16; accepted 2023-05-03