Ripples of a Mutation — An Empirical Study of Propagation Effects in Mutation Testing

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

The mechanics of how a fault reveals itself as a test failure is of keen interest to software researchers and practitioners alike. An improved understanding of how faults translate to failures can guide improvements in broad facets of software testing, ranging from test suite design to automated program repair, which are premised on the understanding that the presence of faults would alter some test executions.

In this work, we study such effects by mutations, as applicable in mutation testing. Mutation testing enables the generation of a large corpus of faults; thereby harvesting a large pool of mutated test runs for analysis. Specifically, we analyze more than 1.1 million mutated test runs to study if and how the underlying mutations induce infections that propagate their way to observable failures.

We adopt a broad-spectrum approach to analyze such a large pool of mutated runs. For every mutated test run, we are able to determine: (a) if the mutation induced a state infection; (b) if the infection propagated through the end of the test run; and (c) if the test failed in the presence of a propagated infection.

By examining such infection-, propagation- and revealability-effects for more than 43,000 mutations executed across 1.1 million test runs we are able to arrive at some surprising findings. Our results find that once state infection is observed, propagation is frequently detected; however, a propagated infection does not always reveal itself as a test failure. We also find that a significant portion of survived mutants in our study could have been killed by observing propagated state infections that were left undetected. Finally, we also find that different mutation operators can demonstrate substantial differences in their specific impacts on the execution-to-failure ripples of the resulting mutations.

CCS CONCEPTS

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

KEYWORDS

software fault infection, error propagation, mutation testing, dynamic analysis, empirical study

ACM Reference Format:
Hang Du, Vijay Krishna Palepu, and James A. Jones. 2024. Ripples of a Mutation — An Empirical Study of Propagation Effects in Mutation Testing. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639179

1 INTRODUCTION

The mechanisms of how software faults lead to observable failures are of obvious interest to practitioners and researchers. For practitioners, such fault-to-failure effects can inform their debugging efforts and can inform their design choices for testing and error recovery. For researchers, such fault-to-failure effects can inform their innovations in fields such as testing, fault localization, and automatic program repair. In this work, we describe an empirical analysis of faults (as mutations) and track their runtime effects at multiple points of their execution to better and more extensively describe the ways in which they may (or may not) lead to failure.

Early research into the runtime effects of faults described the process by which faults lead to software failure. Morell and Offutt, *et al.* [14, 38, 40] described the process by which software faults cause software failure through **R**eachability (*i.e.,* execution), **I**nfection, and then **P**ropagation, which was later named the RIP model of software failure. Voas [47] also describes this phenomenon with the PIE model (*i.e.,* **E**xecution, **I**nfection, and **P**ropagation, acronym in reverse). More recently, the RIP model was extended to include a final requirement for faults to cause failure, which includes **R**evealability — to form the new RIPR model of software failure [4, 27].

With these models in mind, several researchers (*e.g.,* [2, 5, 12, 15, 21, 28, 35, 36, 41, 42, 48, 50]) performed targeted empirical studies to better understand how parts of these models describe how faults lead to failures. Of particular interest, a number of researchers [2, 21, 28, 34–36] investigated the related phenomena of *coincidental correctness* (CC) and **F**ailed **E**rror **P**ropagation (FEP). Both CC and FEP describe a runtime effect of executing a fault, but that fault somehow does not cause an observable failure. Such phenomena are of particular interest for researchers who are creating fault-localization techniques [5, 20, 21, 44], because the fault execution and infection of the state do not correspond to software failure, and hence such localization techniques may be confused or misled. Although valuable in their insights, the research focuses on those executions that lead to non-failure excludes many other runtime effects, for example, the various ways in which execution of faults can lead to failure.

While such targeted investigations are valuable to our understanding of how infection can sometimes not lead to failure, in this work we seek to extend the scope of study from end-to-end —

from R (Reachability) to R (Revealability) in the RIPR model — and also better describe the ways in which various factors, such as the program or fault type, influence those effects. We study not only the ways that fault execution leads to non-failure (*i.e.,* CC and FEP), but instead study *all* phenomena that occur upon fault execution, in an attempt to better understand the full range of software behaviors and the prevalence of each. Moreover, in this work, we extend the size of such studies, in terms of the number of software projects and faults, to get a more complete picture of such phenomena.

To achieve this detailed and more comprehensive study, we utilized mutation testing to inject over 40,000 mutation-based faults and developed an empirical framework that allows for generalizable and reproducible investigations of the end-to-end execution of faults, their infections of the state, the propagation of those infections, and the revealability of those infection propagations. We describe our empirical framework to assess various aspects of the RIPR model, including similar investigations into coincidental correctness and FEP of prior work, but also extending such investigatory scopes to assess other paths for a fault execution to possibly infect state, possibly propagate infection, and possibly be revealed. Such diverse possible runtime effects can be described as *ripples* of the fault, *i.e.,* the many ways in which Reachability (*i.e.,* execution) of a fault may (or may not) lead to Revealability (*i.e.,* failure). Moreover, our empirical framework also allows for several other insights that can be gleaned through our analysis.

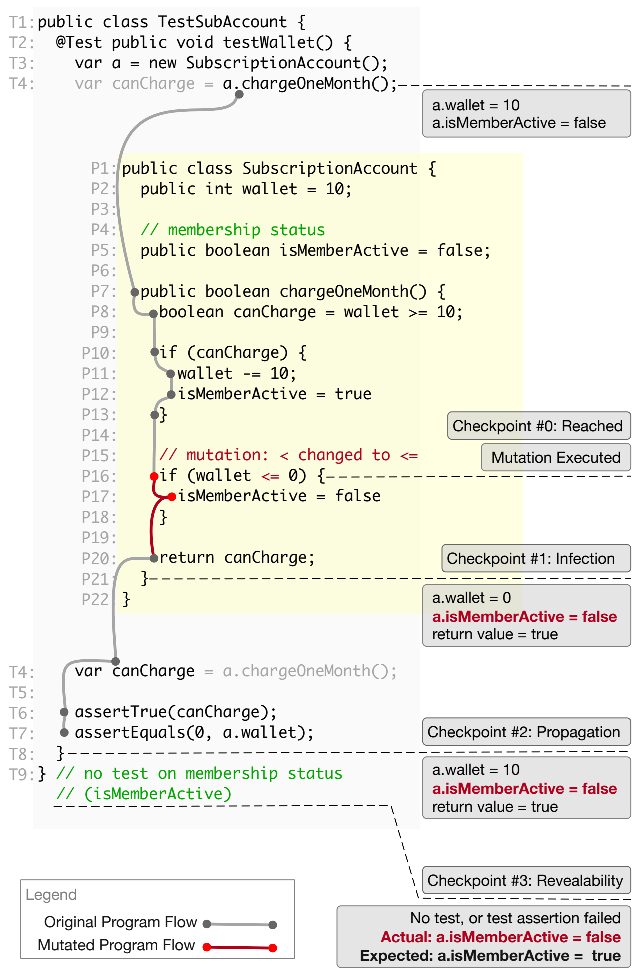
As a result of our study, we found a number of revealing results that should give concern to the fault-based software testing community, or at least, these results should be taken into consideration in related future work. We found that state infections typically follow mutation execution, with infection rates ranging from 64.3% to 94.1%, and propagation rates ranging from 84.9% to 92.6%, given infection. However, there is a noticeable disparity between propagation and test revealing. Moreover, infection usually exhibits as method exit anomalies, such as uncaught exceptions, thrown within the mutated method and leads to early termination of mutation test runs, leaving test oracles unchecked. Furthermore, approximately 18.5%–89.4% of surviving mutants could potentially be killed based on infection or propagation information. Finally, different mutation operators can demonstrate startlingly different impacts in specific RIPR stages.

The main contribution of this paper can be summarized as:

- We propose and implement a practical and scalable end-to-end RIPR analysis framework for mutants. The experimental data are open sourced[1] and available for replication.
- We studied 10 popular open-source projects, performing RIPR analysis on over 40,000 mutants, and executing over 1,000,000 mutation test runs.
- We unveiled several revealing results across our subject programs through our comprehensive RIPR analysis, including (1) a recurring trend in the execution-to-failure ripples of mutation executions, (2) a detailed investigation into unexpected exceptions and the corresponding early-termination phenomenon triggered by mutation, (3) an estimation of killable surviving mutants based on the existing test suite, and (4) the ripple variances led by different mutation operators.

[1] https://doi.org/10.5281/zenodo.10505175

2 BACKGROUND AND MOTIVATION



```
public class TestSubAccount {
  @Test public void testWallet() {
    var a = new SubscriptionAccount();
    var canCharge = a.chargeOneMonth();

public class SubscriptionAccount {
  public int wallet = 10;

  // membership status
  public boolean isMemberActive = false;

  public boolean chargeOneMonth() {
    boolean canCharge = wallet >= 10;

    if (canCharge) {
      wallet -= 10;
      isMemberActive = true
    }

    // mutation: < changed to <=
    if (wallet <= 0) {
      isMemberActive = false
    }

    return canCharge;
  }
}

    var canCharge = a.chargeOneMonth();

    assertTrue(canCharge);
    assertEquals(0, a.wallet);
  }
} // no test on membership status
  // (isMemberActive)
```

Figure 1: Illustrating the flow of a mutated program's passing test run, modeled using RIPR: Reach, Infect, Propagate, Reveal.

To fail on a test assertion, a bug's effects must ripple through multiple stages of a program's test execution. First, the test run must reach the bug. Upon execution, the bug must cause an infection in the program's runtime state. This state infection may infect other parts of the program's state, thus propagating itself through the program's test execution. The infected state must propagate enough, such that it is accessible within the scope of the test case method and its test assertions. To ultimately trigger a failure, the test's assertion would need to detect a deviation in the program's (infectious) output from its expected result. We think of this as the RIPR model of a software fault's execution: "**R**each → **I**nfect → **P**ropagate → **R**eveal" — as originally proposed by Li and Offutt [27].

But do faults always cause such rippling effects from execution of the fault (*i.e.,* reach) to the exposed external symptom of the infection (*i.e.,* reveal)? To better illustrate the RIPR model of analyzing faults, we use a minimal code example as shown in Figure 1.

***Motivating Example for* RIPPR.** Figure 1 depicts the execution flow within a test run of a mutated program. The execution flow starts out with the test method `testWallet` (line T2), where it creates a new account object of the type `SubscriptionAccount`, and invokes the `chargeOneMonth` method on the account. This method invocation causes the execution to leave the test code and enter `chargeOneMonth` in the project production code, where it runs through a preliminary check on the `wallet` amount, to then process monthly subscription fees. Upon charging the monthly fees, the execution sets the `isMemberActive` to true.

Reachability. However, the test run proceeds to execute a mutation on line P16 next, that changes a "less than zero" (`< 0`) conditional check on the `wallet` amount to a "less than or equals zero" (`<= 0`) check. This is a case where the test execution has managed to reach a fault in the program — demarcated as "Checkpoint #0: Reached" in Figure 1.

Infection. Executing the fault causes the `isMemberActive` method to be erroneously reset to false, since the value of `wallet` at this point in the execution is 0. This infection persists to the end of the `chargeOneMonth` execution, and escapes the runtime scope of the mutated method — highlighted as "Checkpoint #1: Infection" in Figure 1.

Propagation. As the test execution returns back to the test code to line T4, the infection in `isMemberActive` returns with it. The test runs assertions on `chargeOneMonth`'s return value and the value of `wallet`, which it expects to be 0. Figure 1 shows a signpost for "Checkpoint #2: Propagation" at the end of the test method, highlighting the infected state for `isMemberActive`. We consider this as an example of an infection propagating through a whole test run.

Revealability. However, notice the lack of any assertion for the value of `isMemberActive`. A lack of any validation for `isMemberActive` causes the test to pass — even as there is an erroneous, or infected program state that has propagated to the test code. In other words, the test method `testWallet` fails to reveal faulty logic that updates the `isMemberActive` property of the `SubscriptionAccount`, even when the test executed the fault. We depict this lack of revelation in Figure 1 with "Checkpoint #3," where we show both the actual and expected state of `isMemberActive`.

Such an example motivates us to consider the mechanics of how faults are revealed (or not revealed) as failures. Following the four "Checkpoints" depicted in Figure 1, we ask such questions, as: Upon execution, how consistently do faults produce an infection? How many such infections propagate through test executions? Finally, how many such propagated infections are revealed as test failures?

Prior Work. Existing works have investigated these effects for real software bugs [2, 11, 21, 48], manually seeded artificial bugs [28, 33–36, 51], and to a certain extent for automatically generated mutations [5, 21, 27, 48, 50]. Such works were typically motivated by the goal of improving the effectiveness of spectra-based fault localization techniques, where the sensitivity and precision of test cases to detect faults is vital. However, such works often examined only a limited number of faults (real or artificial), for a small set of programs. Furthermore, prior studies did not investigate all effects — `Infection`, `Propagation` and `Reveal` — for the faults they considered. Propagation is either considered within the (mutated) statement level or approximated as test failures. In other words, prior works were selective in the effects that they studied for the limited number of faults in their experiments.

Our Investigation. In this work, we study the `RIPR` effects specifically for mutations — localized, artificially generated software faults — as applicable in mutation testing. Limited attention has been paid to the `RIPR` effects of executing mutations, in part due to the scale of the number of mutations generated by mutation testing frameworks. However, we are motivated by such large volumes of mutations offered by mutation testing. We believe that the experimental data from such large volumes mutation test runs would lend confidence in the trends we observe, and in the conclusions we derive from them. Indeed, we collect data about the `RIPR` effects for nearly all auto-generated mutations in their respective test runs, for the programs in our study.

Further, we are not selective about the effects that we study; *i.e.,* for every mutation in our experiments, we examine the `Infection`, `Propagation` and `Reveal` effects, for every test case that reaches and executes the mutation. This would enable our experimental analysis to evaluate possible correlations between such different effects. For instance, using our data we may answer questions such as, *"how often do mutations produce any infection?"*; or *"how consistently do infections propagate to the test code?"*; or *"how often do tests detect propagated infections, revealing as test failures?"* Answering such questions about how mutation-caused infections occur, propagate and get revealed as test failures has practical implications for both mutation testing and improving tests in general.

With such research and results, we may better understand such rippling effects of fault execution, and thus better inform our work as developers and researchers. For example, quantifying the number of mutations that do not produce an infection in the program's state can help estimate equivalence of mutants with the original program. Understanding the degree of infection- and propagation-effects caused by mutations, and any correlations between them, can guide a programmer's attention to mutations (and mutation operators) that are more likely to cause infections that propagate as program outputs. Finally, undetected state infections that propagate back to the test code present clear potential for additional tests and test assertions that detect such uncaught infections to cause failures.

3 METHODOLOGY: SCALABLE ANALYSIS OF MUTATED TEST RUNS

Devising an experiment for an empirical study that analyzed millions of mutation test runs posed unique challenges. To compare the `RIPR`-effects of different mutation runs, our experiment required a generic analysis to compare executions for any two mutation test runs. In this section, we describe the key concepts of such an execution analysis, and also outline the steps in that analysis, which employ those concepts. The step-wise application of these concepts allowed us to implement an experimental setup that evaluated and compared over 1.1 million mutation runs and 11 million no-mutation test runs.

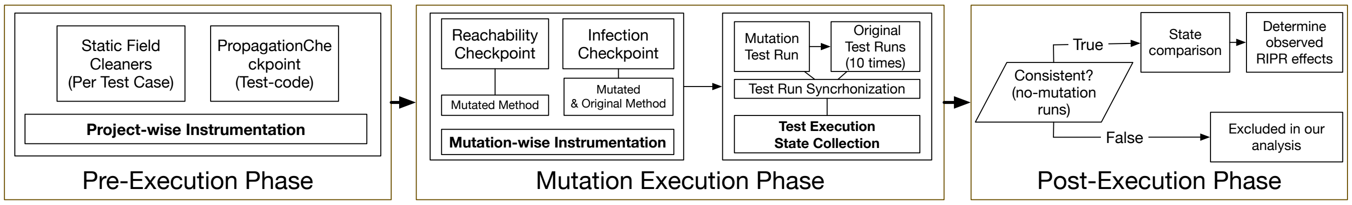

Static Field Cleaners (Per Test Case)
PropagationCheckpoint (Test-code)
Project-wise Instrumentation
Pre-Execution Phase
Reachability Checkpoint
Infection Checkpoint
Mutated Method
Mutated & Original Method
Mutation-wise Instrumentation
Mutation Test Run
Original Test Runs (10 times)
Test Run Syncrhonization
Test Execution State Collection
Mutation Execution Phase
Consistent? (no-mutation runs)
True
False
State comparison
Determine observed RIPR effects
Excluded in our analysis
Post-Execution Phase

Figure 2: Steps in the Scalable Analysis of Mutated Test Runs

3.1 Key Concepts

3.1.1 Using Mutations to Create Faults. Our methodology uses mutation operators, as applicable in mutation testing, to inject faults in real-world subject programs. Using mutation operators allows us to create a vast corpus of software faults. Such faults are localized to individual instructions within the original programs. The generated mutations, when executed by their respective test cases, offered over 1.1 million mutated test runs for analysis.

3.1.2 Synchronizing the Original and Mutated Test Runs. Localized mutations to the original program may potentially change execution flows when compared to the original run. While expected, such mutation-induced deviation in execution flow can potentially cause the mutated run to be drastically different in terms of program semantics and outcomes, but also with regards to changes in the granular data- and control-flows. Comparing such differing executions, even for the same program may not always lend itself to scalable automation.

Figure 1 shows an example deviation in the execution flow when a mutation on line P16 is executed. Executing the mutation on line P16, causes the execution to enter the if-block on line P17 (shown in the color red), and this resets the value of `isMemberActive` to `false`. However, without the mutation on line P16, the execution would have skipped the if-block entirely, causing the execution to jump from line P16 to P20, and this leaving `isMemberActive` to be `true`. We approach this by selecting execution checkpoints that are comparable across the original and mutated test runs, despite differences in the two runs. At such comparable execution checkpoints, across the mutated and original runs, we check for differences in the program's runtime state. The selection of such checkpoints becomes a key concern, and we describe them next.

3.1.3 Selecting Comparable Checkpoints. We adopt the RIPR model, to select comparable checkpoints across the original and mutated runs, and thus synchronize the executions across such checkpoints. Specifically, for every execution (mutated or original), we identify four comparable checkpoints at the points of: (a) Reachability, (b) Infection, (c) Propagation, and (d) Revealability.

Reachability. We define the "Reachability Checkpoint" as the point in the program's execution that executes the mutation for the first time. Figure 1 depicts this as "Checkpoint #0" in our motivating example. Indeed, the mutation will only be executed in the mutated run, and not the original run of the program. As such, to identify a comparable point in the original run, as the "Reachability Checkpoint," we adopt the following approach:

(1) We note the enclosing method that contains the mutation; we refer to that method as M;
(2) We then trigger the mutation run, and count the number of times method M was executed before the enclosed mutation is executed for the first time; we refer to that count as i, and denote M_i as the iteration of M that first executed the mutation.
(3) We then run the original program, and mark the i^{th} execution of M as the "Reachability Checkpoint" in the original run. Note that M would be executed without the mutation in the original run.

We adopt this count-based approach for the "Reachability Checkpoint" because a method that contains a mutation may need to be executed more than once before the enclosed mutation is executed. As such, simply selecting the very first execution of the method in the original test, may not be an appropriately synchronized checkpoint when compared with the mutated run in which we can detect the exact execution instance of the mutation.

Infection. We define the "Infection Checkpoint" as the method-exit of the mutation's enclosing method (M), after the mutation's first execution. Again, the original test run will not have the mutation for us to track its execution. As such, we will first identify M_i (*i.e.,* the "Reachability Checkpoint," as stated above), and mark its method exit as the "Infection Checkpoint" in the original test run. Further, when tracking method exits in both original and mutated runs, we not only account for graceful exits via return statements, but also ungraceful exits via unhandled exceptions.

Prior works have defined the location of the "infection checkpoint" in different ways, *e.g.,* the program expression that contains the mutation [22, 25, 41]; or within the mutant's enclosing statement [22, 41]; or at the end of the mutation's basic block [22, 41]; or before the return statement [2, 21]; or outside the method containing the mutation [2, 21].

Unlike such prior works, our infection point definition at the end of the method's execution allows us to compare program state (*e.g.,* from local variables and fields) that was used throughout the method's scope, and not be limited to the state available in inner scopes of an individual basic block, statement, or expression. Moreover, it offers a stronger signal of state infection because it requires that such infection survives to the exit of the method.

Propagation. For both mutated and original runs, we mark the method-exit of the test case method as the "Propagation Checkpoint." Such an execution point would mark the end of the test case. When comparing the state differences at the "Propagation Checkpoint," across a mutated and original run, we compare all state (*e.g.,* return values, object instantiations, fields, static/global variables) that is accessible from the test case method in question.

Figure 1 depicts the "Propagation Checkpoint" as "Checkpoint #2" for our motivating example's test flow.

Revealability. Finally, we use the result of the test case run, *i.e.,* pass or fail, to compare and synchronize the "Revealability" of the mutated and original test runs. In Figure 1, we show this as "Checkpoint #3." In our motivating example, the mutated test run passes, just as with the original run; we thus would consider the mutation at line P16 as unrevealed by the test method `testWallet`.

3.1.4 *Comparing Program States.*

Figure 1 depicts examples of runtime program state that we monitor and compare across such checkpoints (from the original, non-mutated test run with the mutated test run), which may include values contained within local variables, fields of an object that is accessible from the local state, and shared state such as globally declared static variables. Further, when observing state differences, we account for both primitive values and entire object graphs for user-defined types.

However, comparing granular primitive values or object graph structures at multiple execution points, for over a million test runs would not have been a scalable approach. Instead, we perform hashing (using the SHA-265 hashing algorithm) of the collective program state observed at such synchronized checkpoints. We then used differences in such computed hashes to ultimately detect differences in runtime states at synchronized checkpoints across original and mutated test runs.

Additionally, program state can often be different across two different executions due to non-deterministic factors that are independent of any fault or mutation. Such examples include, but are not limited to, values produced by pseudo-random generators, or ephemeral object identifiers (*e.g.,* hashCode in Java) that change across program runs. We detect such non-determinism in program state and exclude it before comparing execution states. We do this by running the original program's test runs several times and creating an exclusion list of fields and variables that are known to have shown differing runtime values across the several runs, and excluding their values at the time of computing hashes for the observed program state.

3.1.5 *Running Test Cleaners.*

Finally, we would also note that between test runs, we employ cleaner code (via test code instrumentation) that would clear out any state pollution taking place by prior test runs. In our early pilot studies, we found several instances of state pollution, especially in shared global variables, which were also investigated in flaky test research [31, 45]. Employing test cleaners would mitigate flakiness led by test-order dependencies introduced by mutations [45]. Further, as noted above in the discussion on capturing program state, we run the original test run several times, where we check against any accidental flakiness in the original test run.

3.2 Detailed Steps within the Analysis

The aforementioned key concepts are applied in a multi-step process to analyze the mechanics of how mutations are revealed as failures by tests, as shown in Figure 2. In particular, analyzing the RIPR-effects of mutations within a single software project involves the following three phases, which we describe below: (a) a pre-execution phase; (b) a mutation-execution phase; (c) a post-execution phase.

3.2.1 *Pre-Execution Phase.*

Before executing any mutated or original test runs, the pre-execution phase instruments the entire project with three different kinds of probes to perform different tasks:

- *Static Field Cleaners.* These probes are inserted in the test code, to clear out any polluted state in the program's globally shared variables, *e.g.,* static fields. Such state pollution may take place when executing a test, and may cause flakiness and non-determinism in subsequent test runs.
- *Propagation Checkpoint Probes.* These probes are inserted in the test code, and are designed to monitor and dump program state that is accessible from the test code as the test finishes, at the "Propagation Checkpoint."

3.2.2 *Mutation-Execution Phase.*

After instrumenting the program and its test suite with an initial set of probes, the analysis proceeds to execute the test cases against the mutations and the original program. Each mutation warrants its own set of probes to collect program states at the synchronized checkpoints from the original and mutated test runs. For each mutation, the program is instrumented with the following probes in the specific method that contains that specific mutation:

- *Reachability Checkpoint Probe.* This instrumentation probe identifies the first time the mutation was executed. Additionally, it tracks the number of times the mutation's method was invoked for the mutation to be executed the first time. The count of method invocation from the mutated run is also used to identify the corresponding "Reachability Checkpoint" in the original program run.
- *Infection Checkpoint Probe.* This instrumentation probe collects program state at the "Infection Checkpoint." These probes are placed at the method-exit points and are activated after the mutated-, or original-execution arrives at the "Reachability Checkpoint."

After inserting probes for a specific mutation, the analysis proceeds to execute tests against the mutation, and collects runtime states at the infection and propagation checkpoints in the mutated and original test runs. The mutated run is performed first, followed by several rounds (10 times in our experiment) of the original test runs. The multiple rounds of the original test runs help guard against mistakenly treating non-deterministic runtime state as possible infection, and flaky execution flows (as discussed in Section 3.1.4). Once the current mutation's test runs are analyzed, its corresponding instrumentation is completely removed, in preparation for analyzing the next mutation.

3.2.3 *Post-Execution Phase.*

In the final phase of the analysis, the program state from the original test runs, which were executed ten times, are analyzed to detect flaky test runs. If a test run for the original program (*i.e.,* without the mutation) exhibits flaky execution flows across its ten iterations, then we exclude the results of that particular test, for that specific mutation. If however the ten test iterations were consistent, then the test results, specifically the captured runtime states are compared across the original and mutated test runs.

Additionally, these ten test iterations are also used to create an exclusion list of fields that store non-deterministic states (*e.g.,* as a result of pseudo-random generators). Note that such non-deterministic, differing values may show up across test runs that are otherwise non-flaky and consistent in their test (pass/fail) results.

4 EMPIRICAL STUDY

In this section, we outline the empirical design to answer key research questions to unveil the ripples of mutations. First, we describe the way in which we operationalize the methodology that we presented in Section 3. Then, we provide subject programs for our empirical study. Finally, we enumerate our research questions that direct our study, along with rationale and empirical measures that we used to investigate them.

4.1 Implemented Empirical Methodology

To actualize our empirical methodology described in Section 3, we performed our experimentation on Java programs (see Section 4.2), using mutations to allow us to scale up to tens of thousands of faults. For mutation, we use the PIT [10] mutation-testing tool. We modified PIT's source code to allow us to customize and extend its functionality. Our PIT extensions enable a number of features that are needed for our methodology: it (1) performs per-mutation instrumentation of the mutated run to identify the exact method-execution instance when the mutation was first executed, (2) perform per-mutation instrumentation on both the mutated and original, non-mutated program to output state information at the infection checkpoint (*i.e.,* the exact method-execution instance identified by the prior point), and (3) allows us to run the original non-mutated program several times (in our experiment, we ran it ten times) to identify any state that were non-deterministic (*e.g.,* hash codes, thread IDs) so that we could later exclude them from the state comparisons.

We wrote customized instrumentation using the ASM [7] Java class manipulation tool. Our instrumentation injects probes into the program to identify the point in the execution at which the mutation was first executed and log all state values. To output all of the state, we used the XStream library [1].

For the first (reachability) checkpoint, the probes are placed immediately before the mutation to log the occurrence of the mutation execution. For the second (infection) checkpoint, the probes are placed on all method exits of the mutated method and log the memory state upon the first method-execution instance in which the mutation was executed. For the purposes of our study, we define "infection" as any differences found between the mutated and original non-mutated program run in the memory state (excluding those that we identify as non-deterministic, as described above) or method-exit behavior (*e.g.,* original test run exits with a `return` instruction, but the mutated run exits with an `athrow`). For the third (propagation) checkpoint, the probes log accessible memory state (*i.e.,* return values, static fields, object instantiations) within the test (*i.e.,* JUnit) code. For the purposes of our study, we define "propagation" as any differences found between the mutated and original non-mutated program run in the memory state (excluding those that we identify as non-deterministic, as described above) and/or uncaught exceptions that lead to test failure. Finally, for the fourth (revealability) checkpoint, we log the test outcome of pass or fail. We monitor for flaky tests by way of the repeated non-mutated runs and exclude any tests that fail for the non-mutated run and/or exhibit flakiness. The proportion of such exclusion is reported in Table 1. Additionally, we monitor the number of test assertions that are executed within the test code for each test case.

Table 1: Experimental Subject Programs

Subject Project	KLoC	MS	#T	avg. #RT	%Ex	#Run	#Mut
commons-cli	6.2	90%	438	48.1	0.4%	35,940	719
joda-money	9.2	80%	1,483	104.3	0.6%	97,957	895
cdk-data	10.6	46%	4,429	69.7	3.3%	213,972	1,371
jline-reader	13.3	46%	189	30.3	4.1%	125,451	2,808
commons-valid.	16.8	72%	533	17.2	2.7%	35,632	1,725
commons-codec	24.1	85%	1,156	10.7	19.9%	33,766	3,286
spotify-web-api	24.4	60%	291	7.7	1.1%	20,802	1,920
commons-text	26.6	85%	1,242	9.7	4.7%	34,266	3,784
dyn4j	66.1	77%	2,348	28.2	12.7%	249,480	8,143
jfreechart	138.4	33%	2,344	9.5	5.3%	320,357	18,926

4.2 Subject Programs

Our experiments were performed on ten open-source Java projects, as detailed in Table 1. We selected real-world Java projects built with MAVEN, with JUNIT5 test cases[2] written by developers, and whose source code is accessible on GitHub.

All chosen subjects minimally rely on mocking libraries, are not heavy on threads, support XStream serialization, and do not have tests explicitly tagged as flaky tests. Each column of Table 1 presents the following: (1) the subject project, (2) lines of code (KLoC)[3], (3) mutation score (MS), (4) number of test cases (#T), (5) average number of reaching tests (avg. #RT) for each mutation, (6) the proportion of test runs excluded (%Ex) from our analysis, (7) number of analyzed test runs (#Run), and (8) number of analyzed mutants (#Mut). In all, we analyzed execution data from over 1.1 million mutated test runs, including another 11 million non-mutated test runs (to distinguish deterministic state from non-deterministic, as well as to control for test flakiness), for 43,577 mutations. The scale of this analysis required over a week of computational time (on a 3.2Ghz ARM-based M1 processor with 16 GB of RAM).

4.3 Research Questions

In an effort to comprehensively understand the rippling effects of fault infection and propagation, we devised the following set of key research questions:

RQ1 How do ripples of mutations progress?
RQ2 How does mutation execution affect test execution behaviors?
RQ3 What is the potential number of surviving mutants that could have been killed?
RQ4 How do mutation operators affect ripples of mutations?

RQ1*: How do ripples of mutations progress?* To comprehend the progression of how mutations infect the states, propagate into the test code, and are eventually revealed by test failures — a phenomenon of test progression we term as "ripples" — we comprehensively

[2]We translated any JUnit4 test cases to JUnit5.
[3]Lines of code are calculated as the number of Java source code lines using the statistics plugin in Intellij

monitor executions according to our empirical methodology described in Section 3. With the resulting execution data, we can categorize test runs to determine the degree to which mutations ripple through the directed acyclic decision graph, in Figure 3.

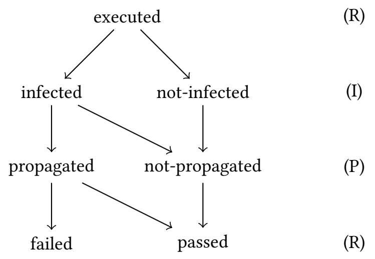


Figure 3: Conceptual Sankey Diagram

RQ2*: How does mutation execution influence test execution behaviors?*

Our initial investigation revolves around the effects of mutation execution on the altered method's exit behaviors. Existing research suggests that mutations can cause test failures due to uncaught exceptions, with some of these instances subsequently classified as **trivial** [23]. A significant proportion of failing test runs involved in mutation analysis result from exogenous crashes (*i.e.,* exceptions triggered by the Java virtual machine or third-party libraries) and source-code oracles (*e.g.,* defensive programming) [16]. We conjecture that such test failures, not being revealed by test oracles, are likely detected sooner at our infection observation point—in other words, prior to the exit of the mutated method. For instance, the execution of a mutation might cause an uncaught exception to be thrown from the altered method, which was expected to return a value upon the exit of the original method.

RQ3*: What is the potential number of surviving mutants that could have been killed?* For the test runs that demonstrate execution, infection, propagation, but then surprisingly produce test passes, we turn our attention toward a practical application of this analysis. This involves estimating the number of potentially "killable" surviving mutants, leveraging propagation and infection information. More specifically, our objective is to estimate the quantity of surviving mutants that could, in theory, be killed using state-based propagation and infection data with the existing test suite. As a corollary, this investigation implies the possible maximum number of test-equivalent mutants among surviving mutants.

RQ4*: How do mutation operators affect ripples of mutations?*

Lastly, we investigate the variations in ripples caused by different mutation operators. To answer this research question, we aggregate our results by the mutation type to investigate how differing types of faults could lead to different rippling execution effects.

5 RESULTS

RQ1: How do ripples of mutations progress?

Figure 4 employs Sankey diagrams [24] to illustrate the end-to-end flow and progression of execution, infection, propagation, and revealability in the test runs analyzed per subject project. These diagrams represent the proportion of flows through the varying widths of their categories.

Central to our analysis are five distinct phases in the RIPR model, represented in each diagram: starting from the "Execution" phase and progressing through "Reaching," "Infection," "Propagation," and finally "Reveal". The designation of each phase is indicated at the bottom of the corresponding diagram for each Sankey Diagram, with detailed definitions available in Section 4.1.

Each phase is allocated to a specific column within the diagram, potentially subdivided into multiple categories (or steps). For example, the "Start" phase is located in the leftmost column, containing only the "start" category. As another example, the "Revealability" phase, positioned in the rightmost column, comprises two categories: "PS" (pass) and "FL" (fail).

The color coding in our Sankey diagrams serves to differentiate the category statuses as monitored by our instrumentation probes. Categories depicting impact by a mutation are distinctly marked in red, *i.e., Execution* ("E"), *Infection* ("I"), *Propagation* ("P"), and *Failure* ("FL"). Conversely, categories representing no impact due to the mutation are highlighted in green, *i.e., No Infection* ("NI"), *No Propagation* ("NP"), and *Pass* ("PS").

Furthermore, the width of the flows between categories of different phases is indicative of the proportion of test runs transitioning between categories, providing a visual representation of their relative distribution. The percentages adjacent to each category name in the diagram represent the proportion of test runs within a specific phase, summing to 100% across each phase (column).

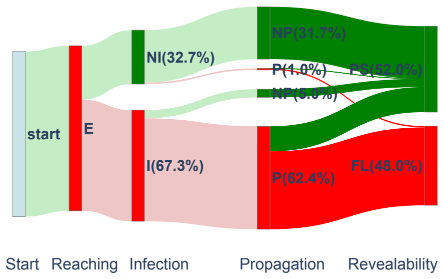
Consider the case of COMMONS-CLI, as illustrated in Figure 4b: All test runs commence in the "Start" phase, followed by the execution of the injected fault[4]. 73.8% of test runs exhibit state infection after its mutation execution. And then 66.2% of (all) test runs show both infection and the propagated infection into test code (which is nearly 90% of the 73.8% of test runs that showed infection). Test runs with observed propagation primarily result in test failures at the revelation stage as computed by the division of the percentages on "FL" and "P" categories.

We find some common mutation ripple patterns (Figure 4) among all subject projects: State infections typically follow mutation execution, with infection rates ranging from 64.3% to 94.1%. Once state infection is observed, propagation is frequently detected, with propagation rates ranging from 84.9% to 92.6% among infected test runs. However, notably, there is a noticeable disparity between propagation and test outcomes. Some test runs pass despite observable propagation in the test code. Test runs showing propagation end up with test failures in 60.3%–95.2% of cases.

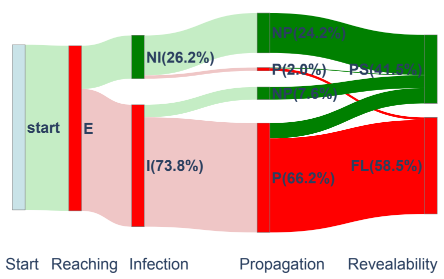
Despite these observations, there are some variances, such as *jline-reader*, where most propagations are not revealed by test runs. Moreover, small offshoots flow from "not infected" to "propagated" in all subjects, accounting for 0.5%–6.8% of all test runs. These flows represent test runs with observed propagation but with no observed infection. This phenomenon occurs due to repeated execution of the mutation — infection may not occur on the first execution of the mutation when our probes check for it, but only infects the state on a subsequent execution of the mutation site.

OBSERVATION 1. *All subject projects exhibit similar mutation-triggered ripples. While infections usually permeate into the test code, they do not necessarily give rise to revelations.*

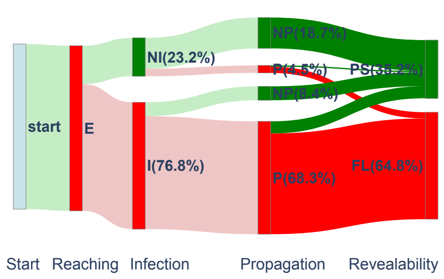
[4]Test runs that fail to reach the injected faults are excluded from this analysis.



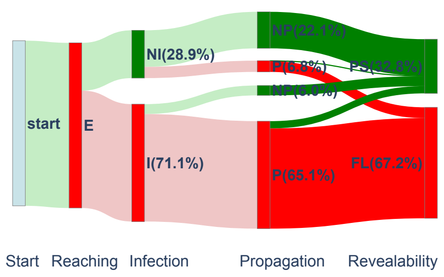
(a) cdk-data



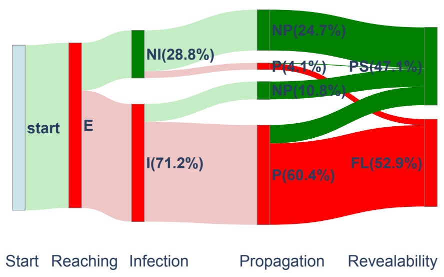
(b) commons-cli



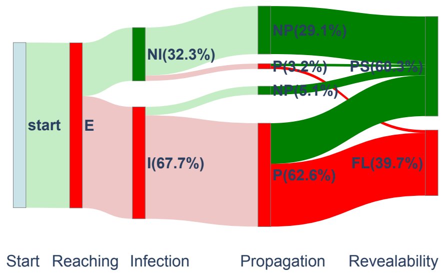
(c) commons-codec



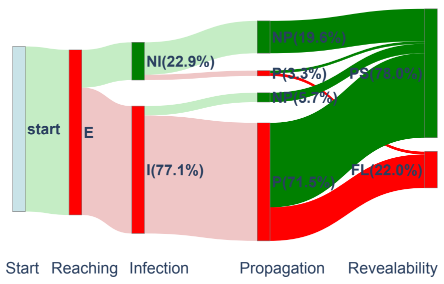
(d) commons-text



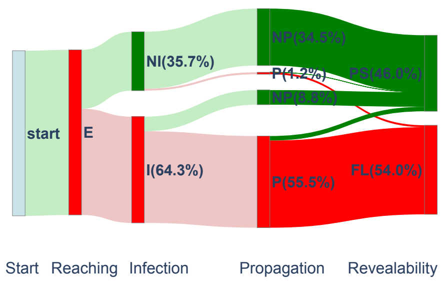
(e) commons-validator



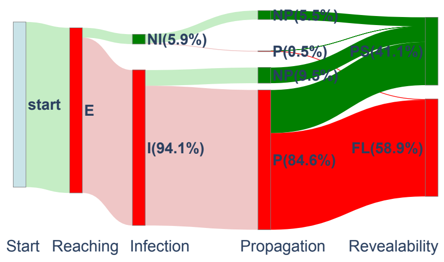
(f) jfreechart



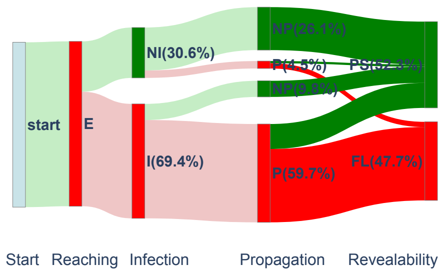
(g) jline-reader



(h) joda-money



(i) spotify-web-api



(j) dyn4j

E: execution I: infected NI: not infected P: propagated NP: not propagated PS: pass FL: fail

Figure 4: Ripples of mutations — Sankey diagram per project

RQ2: How does mutation execution influence test execution behaviors?

Table 2 presents how mutations alter the exit behaviors of mutated methods at our infection observation point. The first column denotes the count and percentage of failing test runs related to non-return mutation operators[5]. The second column denotes the

[5]Return value mutation operators merely mutate the return value without causing any exceptions at method exit, hence related test runs are excluded here.

Table 2: Method Exit Anomalies

Program Name	#Failed	Anomaly	Source-Code Oracle
commons-cli	13,965 (60.32%)	4,216 (30.19%)	1,728 (40.99%)
commons-text	17,832 (67.95%)	4,985 (27.96%)	1,882 (37.75%)
joda-money	36,495 (50.68%)	10,486 (28.73%)	5,931 (56.56%)
jline-reader	21,448 (19.36%)	9,842 (45.89%)	1,529 (15.54%)
commons-validator	13,340 (51.42%)	4,947 (37.08%)	1,159 (23.43%)
cdk-data	80,398 (45.35%)	38,550 (47.95%)	5,717 (14.83%)
spotify-web-api	2,688 (34.19%)	677 (25.19%)	0 (0.00%)
commons-codec	18,097 (64.04%)	5,250 (29.01%)	1,195 (22.76%)
jfreechart	96,626 (37.92%)	31,676 (32.78%)	12,749 (40.25%)
cdk-data	80,398 (45.35%)	38,550 (47.95%)	5,717 (14.83%)
dyn4j	89,603 (44.43%)	47,065 (52.53%)	22,324 (47.43%)

number of test runs witnessing anomalous method exits and their relative ratio among failing test runs, *i.e.,* the first column. The third column lists the number of test runs in which method-exit anomalies caused by source-code oracles [16], namely intended exceptions specified by the developers, are observed, along with their proportion of the second column.

As an example, for *commons-cli*, 13,965 test runs (60.32%) fail, wherein 4,216 (30.19%) display altered method exits at the infection observation point. This may manifest as a change from a "return" exit in the non-mutated test run to an uncaught exception in the mutated test run. Furthermore, source-code oracles contribute to 40.99% of such anomalies.

Our findings indicate that 25.19%–52.53% of failing test runs encounter an anomaly at the method exits during even the first exit of the mutated method post mutation execution. Surprisingly, source-code oracles contribute up to 56.56% of these anomalies, rendering mutants trivially killed. Notably, *spotify-web-api* does not exhibit such anomalies, largely because this project includes high-level APIs that do not depend on the project-defined source-code oracles for intentional exception handling.

Observation 2. *A significant portion of failing test runs (25.19%–52.53%) exhibit method exit anomalies, largely influenced by source code oracles. These mutations often result in early terminations, even before assertion checks.*

RQ3: What is the potential number of surviving mutants that could have been killed?

We introduce two categories of currently surviving mutants that possibly could have been killed, based on our infection or propagation data, using the existing test suite. First, the propagation of memory infection into test code suggests the potential to augment current test cases with new revealing test oracles. Second, memory infection at either the infection or propagation observation point could potentially help inform new test cases. This can be achieved by introducing source-code oracles [16], inline tests [29, 30], using mocking frameworks to monitor the behavior of the object under test, or by creating a new test case that replicates the arguments passed into the mutated method with new assertions.

In Figure 5, we show how existing test data from the project's test suite could potentially kill surviving mutants in our analysis. "Propagation Kill" refers to surviving mutants that can be killed due to observed propagation into the test code. "Propagation or Infection Kill" indicates surviving mutants that could potentially be killed based on either observed infection or propagation. We observe that 18.5%–89.4% of surviving mutants could potentially be killed using existing test data and propagation information (41.3%, on average, survived mutants could be killed in test code, and 51.8%, on average, survived mutants could be killed in either test or production code). With additional infection information, this number could slightly increase to 35.6%–91.6%. The fact that this difference is relatively small between locally detectable infection and test-code propagated infection can be attributed to the fact that most observed infections are propagated into the test code, as demonstrated in the Sankey diagram in response to *RQ1*. We can also infer that 8.4%–64.4% of surviving mutants are *test-equivalent mutants* [22] (*i.e.,* mutants that cannot be detected by their test cases). Additionally, upon breaking down the mutation operators, we discover that most test-equivalent mutants, using existing test data, are generated by the "ConditionalBoundary" mutation operator.

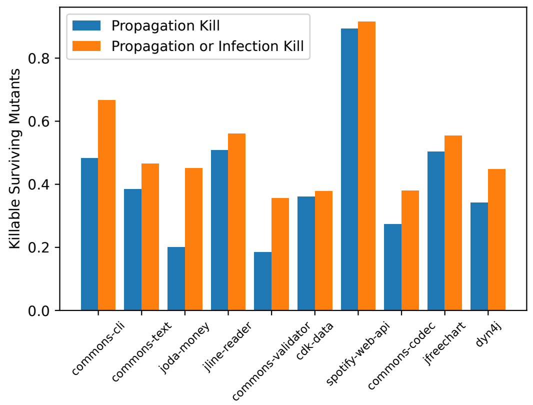


Figure 5: Surviving mutants that are killable by current tests

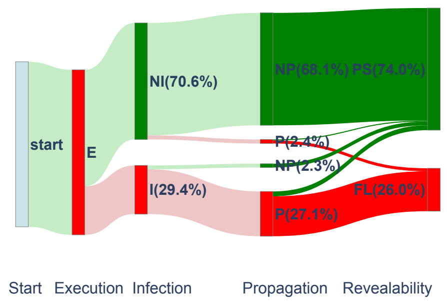
Observation 3. *On average 51.8% of surviving mutants could potentially be killed by existing test suite based on infection and propagation information.*

RQ4: How do mutation operators affect ripples of mutations?

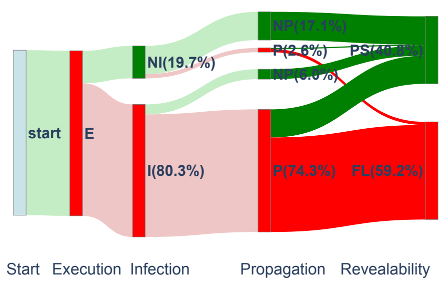
Figure 6 presents Sankey diagrams for six distinct mutation operators, aggregating the top six most frequent test runs across all subject projects [6]. Notwithstanding their shared trait as depicted in Figure 4, *i.e.,* propagation typically follows infection, observable distinctions are found in the specific segments of these mutator-specific Sankey diagrams.

The ripples of certain mutation operators can be inferred from their definitions. Consider, for example, the ripples caused by the *ConditionalBoundary* (Figure 6a) and *NegatedConditional* (Figure 6b) mutation operators. The former is noticeably less adept at inducing state infection (29.4%) than the latter (80.3%), suggesting that the change from "==" to "!=" demands less rigorous test data for

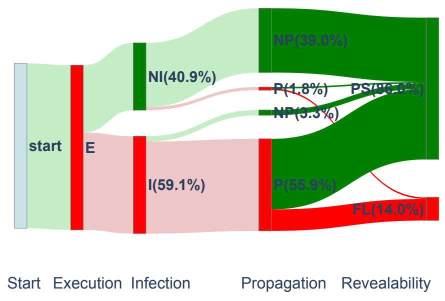
[6]Due to page limitations, the Sankey Diagrams for the other four mutation operators can be accessed in our supplementary material, available at https://doi.org/10.5281/zenodo.10505175



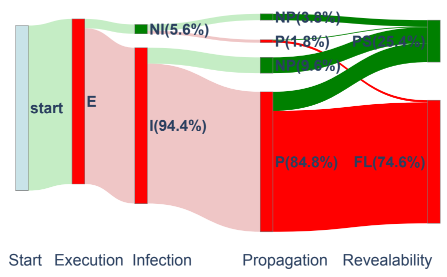
(a) ConditionalBoundary
134505 aggregated test runs



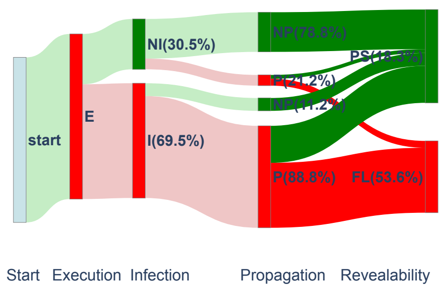
(b) NegateConditional
439264 aggregated test runs



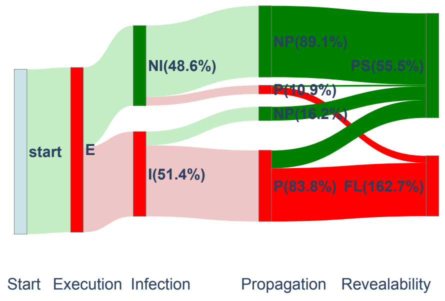
(c) VoidMethodCall
207997 aggregated test runs



(d) NullReturns
97187 aggregated test runs



(e) Math
135194 aggregated test runs



(f) BooleanTrue
44547 aggregated test runs

Figure 6: Sankey diagram aggregated by mutation operators

anomaly detection, while converting ">" to ">=" necessitates more stringent test cases to invoke infection. In the case of the *VoidMethodCall* (Figure 6c) mutation operator, the diagram reveals a prominent divide between propagation and revelation, *i.e.,* only 25.0% of test runs with propagated infection are revealed as test failures, implying that only a minimal subset of test cases possess the required assertions for such propagation being revealed. It can be inferred that *VoidMethodCall* mutation operators, substituting a void method call with a blank one, tend to create state infections often overlooked or checked by a small proportion of test cases by developers. Furthermore, the *NullReturnValues* (Figure 6d) mutation operator, which modifies the return value to null, consistently leads to state infection and propagation, culminating in a significant proportion of test failures compared to the other three operators.

Such heterogeneous characteristics, portrayed through unique segments of the Sankey diagram, suggest that ripples of mutations can differ based on the mutation operators. We further extrapolate that these ripple variances might be applicable to other mutant faults provided by different mutation operators and even real faults.

> Observation 4. *Different mutation operators can demonstrate substantial differences in their specific impacts on the ripples of mutations.*

6 DISCUSSIONS AND IMPLICATIONS

Our large-scale propagation analysis based on the RIPR model unveils unique insights for mutation-based software testing research.

Firstly, propagation cannot be simply approximated as test failures in real-world test cases. Our Sankey Diagrams, as presented in Figure 4 and Figure 6, reveal that infection at the (mutated) method exit often facilitates propagation. However, such propagation is not necessarily revealed by test failures. A point to note in this regard is that we selected the method exit of the mutated method as the point at which we detected infection. A different choice could have been at the instruction that immediately follows the mutated instruction, and such a choice may cause a different likelihood of propagation of the infection back to the test case.

Secondly, the gaps between propagation and test failures indicate the potential for augmenting test suites by leveraging existing test data. Figure 5 shows that a significant portion of surviving mutants can potentially be killed by using infection or propagation information. This can be achieved by adding test assertions, source-code oracles, or inline tests [29, 30], albeit with some code modifications.

Thirdly, a mutant kill usually occurs early, often as a result of an exception being thrown even within the (mutated) method, which could be due to intentional defensive programming by developers. However, if such infection behaviors are deemed undesirable, such as when evaluating the quality of test assertions, mutation operators should be designed with consideration of the mutated method's context to mitigate such immediate, failure-inducing crashes.

Lastly, the choice of mutation operators in mutation-based analysis should be made carefully. Variances in ripples caused by different mutation operators could potentially explain the wide range, 0%–56.4% [21, 34, 50, 51], of previously reported failed-error-propagation or coincidental-correctness rates for mutants or real bugs in Java.

Such coincidental correctness or failed-error propagation may mislead fault localization, but may be preferred by mutation testing, which seeks to identify stubborn mutants, *i.e.,* those that are killable but not easily killed. We posit that the choice of mutation operators can influence the evaluation of fault-localization techniques.

7 THREATS TO VALIDITY

Threats to internal validity primarily stem from our particular choices of observation points and experimental setup for exploring the RIPR model. However, we mitigated the influence of test-order dependencies introduced by mutations [31, 45] and other original test-run inconsistencies at different levels by: (1) repeating 10 original test runs and excluding inconsistent test runs in our analysis, (2) enforcing crafted static-field cleaners per test run, and (3) synchronizing mutation test runs and original test runs with careful instrumentation-point choices

These ripples might manifest differently for other projects or programming languages. However, our experiment's subject programs are mature, open-source projects with human-written test suites. Furthermore, our extensive experimental scale, comprising over 1.1 million mutated test runs in total, offers unique insights into propagation analysis. Finally, we note that findings based on mutations may not necessarily generalize to real-world, human-introduced bugs. However, investigating the rippling effects of mutations is indeed the explicit scoping of this work.

Lastly, threats to construct validity could stem from the mutation testing tool we used. We used PIT with its default group of mutation operators. Though PIT is a commonly-used, mature mutation testing framework [10, 17], there is a risk that a mutation could unintentionally corrupt shared state [45]. To tackle such mutant order dependencies without escalating experimental overhead, we apply patches to potentially corrupted static fields defined in the project before each test run, which we additionally validate by dumping these shared states at the beginning of each test run.

8 RELATED WORK

The RIP model, originating from the concurrent dissertations of Morell and Offutt, evolved under two monikers: Propagation, Infection, and Execution (PIE) [39] and Reachability, Necessity, and Sufficiency [14]. Voas subsequently proposed a dynamic failure model for PIE analysis [47], yielding statistical estimates for crucial fault execution characteristics. The software testing community has since widely adopted these RIP or PIE concepts [3]. This traditional RIP model recently received a critical addition, Revealability, which bridges the gap between propagated infection in test case method and test failures [4, 27]. Our works seeks to further our understanding of such fault-to-failure models, when applied to mutations and mutation testing.

Mutation testing aims to assess test suite quality by aggregating revealing results, *i.e.,* test outcomes, into mutants' killability [13]. Particularly, weak mutation testing accelerates the traditional approach by integrating infection results into this measure [19, 32]. Empirical studies often account for infection and revealability into a mutant's status in both weak and strong mutation testing analysis [25, 41, 42, 49]. Moreover, such infection and propagation information can optimize mutation testing by avoiding unnecessary test executions [22, 25] and aid in estimating equivalent mutants [6]. Our empirical analysis compliments such works by offering additional insights into the mechanics of mutation executions, which may improve mutation-testing efficiency.

Researchers have shown interest in instances where executed program faults, do not alter the output nor lead to test failures. These situations, referred to as failed error propagation, strong/weak coincidental correctness, error masking, or fault masking, exhibit slight variations in definition [5, 9, 21, 26, 33, 35, 37, 48]. They have been investigated in particular in relation to coverage-based fault localization (CBFL) as potential threats to traditional CBFL techniques [2, 33–35, 51]. In contrast, our work is motivated in mutation testing that typically favors stubborn mutants with lower killability ratios [8, 18, 43, 46].

Infection and propagation information can also be harnessed to devise new test cases or enhance existing tests. Li *et al.* [27] outlined diverse test oracle strategies, subsequently assessing their efficacy in auto-generated tests. Similarly, Xiong *et al.* [50] introduced the concept of an 'inner oracle', evaluating the potential of these oracles in reducing test suite size during specific test input executions. Liu *et al.* [29, 30] proposed inline tests, and based on infection information, suggested a mutation-driven method to incorporate test-case-specific oracles into production code. Insights from our empirical study can potentially inform such techniques for test generation and enhancement.

9 CONCLUSION

In this work we study the effects of mutations, in how they infect program state that propagates through the mutated run, to be ultimately revealed as an observable failure. The execution-to-failure effects of mutations have received limited attention in prior works in-part due to the large scale at which mutations are generated by mutation testing frameworks. Whereas, our study is motivated precisely by the large pool of faulty test runs that mutation testing offers to analyze. An examination of such a large corpus of faulty test runs stands to lend confidence to our experimental findings. The results of our investigation show that once a mutation causes infection, it is likely to propagate through the end of the test run; however, such propagation of an infection does not always result in its revelation as a test failure. We also find that the choice of mutation operator can have a substantial impact on the execution-to-failure ripple effects of the resulting mutations. Finally, we note that for the mutations for our subject programs, on average 51.8% of surviving mutations could be potentially killed if the executing tests observes a propagated state infection, by means of a test assertion, source-code oracle, or inline tests.

In future work, we seek to further investigate a number of factors, such as how real faults behave in comparison with mutations. Moreover, we will further analyze state differences at a more fine-grained level to not only whether infection propagates, but perhaps the extent to which it does so.

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] 2022. XStream - About XStream. https://x-stream.github.io/ [Accessed 01-08-2023].

[2] Rawad Abou Assi, Chadi Trad, Marwan Maalouf, and Wes Masri. 2019. Coincidental correctness in the Defects4J benchmark. *Software Testing, Verification and Reliability* 29, 3 (2019), e1696. https://doi.org/10.1002/stvr.1696 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1696 e1696 STVR-18-0045.R2.

[3] Paul Ammann and Jeff Offutt. 2008. *Introduction to software testing*. Cambridge University Press.

[4] P. Ammann and J. Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press. https://books.google.com/books?id=58LeDQAAQBAJ

[5] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M Hierons, and Mark Harman. 2014. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings of the 36th international conference on software engineering*. 573–583.

[6] Amani Ayad, Imen Marsit, JiMeng Loh, Mohamed Nazih Omri, and Ali Mili. 2019. Estimating the Number of Equivalent Mutants. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 112–121. https://doi.org/10.1109/ICSTW.2019.00039

[7] E. Bruneton, R. Lenglet, and T. Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*.

[8] 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

[9] David Clark and Robert M Hierons. 2012. Squeeziness: An information theoretic measure for avoiding fault masking. *Inform. Process. Lett.* 112, 8-9 (2012), 335–340.

[10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[11] Murial Daran and Pascale Thévenod-Fosse. 1996. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes* 21, 3 (1996), 158–171.

[12] Vidroha Debroy and W. Eric Wong. 2009. Insights on Fault Interference for Programs with Multiple Bugs. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering (ISSRE '09)*. IEEE Computer Society, Washington, DC, USA, 165–174. https://doi.org/10.1109/ISSRE.2009.14

[13] R.A. DeMillo, R.J. Lipton, and F.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

[14] Richard A. DeMillo and A. Jefferson Offutt. 1991. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Softw. Eng.* 17, 9 (sep 1991), 900–910. https://doi.org/10.1109/32.92910

[15] Nicholas DiGiuseppe and James A. Jones. 2011. Fault Interaction and its Repercussions. In *Proceedings of the International Conference on Software Maintenance*.

[16] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 715–726. https://doi.org/10.1145/3597926.3598090

[17] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[18] 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

[19] W.E. Howden. 1982. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering* SE-8, 4 (1982), 371–379. https://doi.org/10.1109/TSE.1982.235571

[20] Alfredo Ibias and Manuel Núñez. 2021. SqSelect: Automatic assessment of failed error propagation in state-based systems. *Expert Systems with Applications* 174 (2021), 114748.

[21] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2020. An empirical study on failed error propagation in Java programs with real faults. *arXiv preprint arXiv:2011.10787* (2020).

[22] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/2610384.2610388

[23] 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* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/3092703.3092732

[24] Alex BW Kennedy and H Riall Sankey. 1898. The Thermal Efficiency of Steam Engines.. In *Minutes of the Proceedings of the Institution of Civil Engineers*, Vol. 134. Thomas Telford-ICE Virtual Library, 278–312.

[25] Sang-Woon Kim, Yu-Seung Ma, and Yong-Rae Kwon. 2013. Combining weak and strong mutation for a noninterpretive Java mutation system. *Software Testing* 23 (12 2013). https://doi.org/10.1002/stvr.1480

[26] Janusz Laski, Wojciech Szermer, and Piotr Luczycki. 1995. Error masking in computer programs. *Software Testing, Verification and Reliability* 5, 2 (1995), 81–105.

[27] Nan Li and Jeff Offutt. 2017. Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering* 43, 4 (2017), 372–395. https://doi.org/10.1109/TSE.2016.2597136

[28] Yihan Li and Chao Liu. 2012. Using cluster analysis to identify coincidental correctness in fault localization. In *2012 Fourth International Conference on Computational and Information Sciences*. IEEE, 357–360.

[29] Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. 2023. Extracting Inline Tests from Unit Tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1458–1470. https://doi.org/10.1145/3597926.3598149

[30] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2023. Inline Tests. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 57, 13 pages. https://doi.org/10.1145/3551349.3556952

[31] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 643–653.

[32] Brian Marick. 1991. The weak mutation hypothesis. In *Proceedings of the symposium on Testing, analysis, and verification*. 190–199.

[33] Wes Masri and Rawad Abou Assi. 2010. Cleansing test suites from coincidental correctness to enhance fault-localization. In *2010 third international conference on software testing, verification and validation*. IEEE, 165–174.

[34] Wes Masri, Rawad Abou-Assi, Marwa El-Ghali, and Nour Al-Fatairi. 2009. An Empirical Study of the Factors That Reduce the Effectiveness of Coverage-Based Fault Localization. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)* (Chicago, Illinois) *(DEFECTS '09)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/1555860.1555862

[35] Wes Masri and Rawad Abou Assi. 2014. Prevalence of Coincidental Correctness and Mitigation of Its Impact on Fault Localization. *ACM Trans. Softw. Eng. Methodol.* 23, 1, Article 8 (feb 2014), 28 pages. https://doi.org/10.1145/2559932

[36] Yi Miao, Zhenyu Chen, Sihan Li, Zhihong Zhao, and Yuming Zhou. 2012. Identifying Coincidental Correctness for Fault Localization by Clustering Test Cases.. In *SEKE*. 267–272.

[37] Yi Miao, Zhenyu Chen, Sihan Li, Zhihong Zhao, and Yuming Zhou. 2013. A clustering-based strategy to identify coincidental correctness in fault localization. *International Journal of Software Engineering and Knowledge Engineering* 23, 05 (2013), 721–741.

[38] L. J. Morell. 1990. A Theory of Fault-Based Testing. *IEEE Trans. Softw. Eng.* 16, 8 (aug 1990), 844–857. https://doi.org/10.1109/32.57623

[39] Larry J. Morell. 1990. A theory of fault-based testing. *IEEE Transactions on Software Engineering* 16, 8 (1990), 844–857.

[40] Andrew Jefferson Offutt and R. A. Demillo. 1988. *Automatic Test Data Generation*. Ph. D. Dissertation. USA. AAI8904822.

[41] A. Jefferson Offutt and Stephen D. Lee. 1991. How Strong is Weak Mutation?. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Canada) *(TAV4)*. Association for Computing Machinery, New York, NY, USA, 200–213. https://doi.org/10.1145/120807.120826

[42] A Jefferson Offutt and Stephen D Lee. 1994. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering* 20, 5 (1994), 337–344.

[43] 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

[44] Raul Santelices and Mary Jean Harrold. 2011. Applying aggressive propagation-based strategies for testing changes. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 11–20. https://doi.org/10.1109/ICST.2011.46

[45] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/3293882.3330568

[46] 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

[47] J.M. Voas. 1992. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering* 18, 8 (1992), 717–727. https://doi.org/10.1109/32.153381

[48] Xinming Wang, S. C. Cheung, W. K. Chan, and Zhenyu Zhang. 2009. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 45–55.

[49] MR Woodward and K Halewood. 1988. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Workshop on software testing, verification, and analysis*. IEEE Computer Society, 152–153.

[50] Yingfei Xiong, Dan Hao, Lu Zhang, Tao Zhu, Muyao Zhu, and Tian Lan. 2015. Inner Oracles: Input-Specific Assertions on Internal States. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 902–905. https://doi.org/10.1145/2786805.2803204

[51] Xiaozhen Xue, Yulei Pang, and Akbar Siami Namin. 2014. Trimming Test Suites with Coincidentally Correct Test Cases for Enhancing Fault Localizations. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. 239–244. https://doi.org/10.1109/COMPSAC.2014.32