Please fix this mutant: How do developers resolve mutants surfaced during code review?

Goran Petrović
Google Switzerland, GmbH
Zürich, Switzerland
goranpetrovic@google.com

Marko Ivanković
Google Switzerland, GmbH
Zürich, Switzerland
markoi@google.com

Gordon Fraser
University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

René Just
University of Washington
Seattle, USA
rjust@cs.washington.edu

***Abstract*—Mutation testing has been demonstrated to motivate developers to write more tests when presented with undetected, actionable mutants. To facilitate this effect, modern mutation systems aim to generate and surface only actionable mutants—few in numbers but highly valuable to the developer. This requires a deeper understanding of the extent to which developers resolve surfaced mutants and how: If they decide not to resolve an undetected mutant, why not? On the other hand, if they do resolve a mutant, do they simply add a test that detects it, or do they also improve the code?**

In order to answer these questions we compiled and analyzed a dataset of 1,538 merge requests with corresponding mutants surfaced during the code review phase. Our analysis reveals that determining whether a mutant is indeed resolved during code review is actually a non-trivial problem: for 64% of mutants, the mutated code changes as the merge request evolves, requiring dedicated techniques to precisely resurface the same mutants and to discover which of them remain unresolved after a code change. Overall, our analysis demonstrates that 38% of all surfaced mutants are resolved via code changes or test additions. Out of all mutants that are endorsed by a reviewer, 60% are resolved and result in additional tests, code refactorings, and improved documentation. Unresolved, yet endorsed, mutants stem from developers questioning the value of adding tests for surfaced mutants, later resolving mutants in deferred code changes (atomicity of merge requests), and false positives (mutants being resolved by tests not considered when creating the mutants, e.g., in integration test suites).

***Index Terms*—mutation testing, test efficacy, code quality, code review, mutant resolution**

I. INTRODUCTION

Mutation testing research has traditionally focused on the relationship between a test suite's mutant detection ratio and its true efficacy in terms of detecting real faults. *Mutants* are systematically seeded, artificial faults and while they are simpler than real faults [1], empirical evidence shows that mutants are adequate proxies for real faults and that mutant detection is correlated with real-fault detection [2]–[4]. However the number of mutants grows quickly with program size due to the fact that almost every program statement can be mutated in multiple ways. For example, an assignment of the form `a = b + c` may be deleted altogether, or the right-hand-side operand may be mutated to a constant value (e.g., `a = 0`) or a different arithmetic operation (e.g., `a = b - c`).

As a result of the sheer number of possible mutants, practical deployment of mutation testing tends to be more concerned with the actionabililty of mutants when presented to developers as test goals. Consequently, recent research has moved away from studying mutant detection ratios as an adequacy measure to studying the actionability of individual mutants. For example, researchers at Google have pointed out that the vast majority of generated mutants are not actionable in that they are either unsatisfiable test goals or simply not worth satisfying [5], [6]. Surfacing such mutants as program-analysis findings effectively leads to false positives [7]. In response to this challenge, Google's mutation system relies on aggressive mutant suppression to make mutation testing a viable approach in practice, leading to long-term improvements in test quantity and quality, far beyond improvements observed with surfacing code coverage results [8]. Similarly, researchers at Facebook addressed this challenge by using very few mutation operators learned from past bug fixes to generate very few, yet mostly actionable, mutants that can be surfaced to developers [9].

Industrial deployments of mutation testing are promising, but they also present new analysis challenges, such as identifying and surfacing few actionable mutants and consistently resurfacing them as code changes. Furthermore, prior work also suggests, but did not explore in detail, that even surfacing equivalent mutants may be valuable. *Equivalent mutants* cannot be detected by any test (unsatisfiable test goals) but may expose bugs or otherwise undesirable code, and hence could lead to meaningful code improvements. For example, Coles [10] argued in a keynote that some equivalent mutants are valuable because they expose ambiguity in the code. Similarly, Petrović et al. [6] introduced the notion of productive mutants and argued that equivalent mutants can be productive, if they advance code quality, and that non-equivalent mutants can be unproductive, if they represent test goals that lead to undesirable tests.

This paper aims to provide a better understanding of what mutants developers ignore, what mutants they deem actionable, and what actions they use to resolve them. Using a sample of 2,806 mutants, generated and surfaced to developers during code review, we track whether and how these mutants were resolved. This in itself turns out to be challenging: throughout the code review process, both code and tests change and evolve, and identifying just where a mutant should be resurfaced after these changes is a difficult problem. As a result, we implemented an approach to track mutants across changes, and quantitatively and qualitatively analyzed how developers resolve mutants. Specifically, this paper answers the following four research questions:

RQ1 How often do code locations of surfaced mutants change as code evolves? Can these changes be accurately tracked?

RQ2 How often are surfaced mutants resolved?

RQ3 Is mutant resolution associated with changes to both code and tests?

RQ4 What actions do developers take to resolve mutants?

The key results of the quantitative and qualitative analyses are:

RA1 The code location of 64% of all surfaced mutants changes during code review. Tree-based diffing can track such changes, with an accuracy of over 99%.

RA2 Overall, 38% of all surfaced mutants and 60% of please-fix mutants are resolved during code review. Please-fix mutants are resolved more than twice as often as not-useful mutants.

RA3 Merge requests with resolved surfaced mutants have a significantly higher probability of change to both code and tests, compared to merge requests with unresolved or non-surfaced mutants. The effect is stronger for tests than for code.

RA4 The most common action for mutant resolution is additional testing, followed by code refactorings, and in-depth discussion about code relevance (in particular for statement-deletion mutants). The most common reasons for a lack of observed resolution are developers questioning the actionability of a mutant and deferred action (i.e., mutant resolution in a subsequent merge request).

This paper contributes a deeper understanding of mutant resolution in practice, and can inform the development and deployment of future mutation testing systems.

II. Preliminaries

Our study focuses on mutation testing applied during code review. This section provides an overview of the considered use case and defines terms used throughout the paper.

A. Contemporary Code Review

Contemporary code review [11] is a quality assurance technique in which at least one person other than the author of the code manually inspects source code changes before they are merged into the codebase. It is used by companies of all sizes as well as open source projects [12], [13]. A 2018 study found that, in Google's industrial setting, developers are generally happy with the code review process: all 44 surveyed developers agreed that it is valuable [14], and 97% of all developers at Google are happy with the Critique code review UI.

Contemporary code review is always augmented with automated code analysis tools, such as linters, automated test execution, and code coverage analyzers. It helps developers discover faults, missing tests, test failures, compliance issues, and other code health issues early in the development process. Examples of automated code findings include code coverage information, linter errors, and ClangTidy [15] or ErrorProne [16] suggestions. Examples of human comments include suggestions about code design, reuse of libraries, style, and other code improvements.

The code being reviewed often changes during the code review. Developers react to the automated analysis and modify the code, possibly even before a human reviewer has had a chance to see it. If a reviewer leaves comments, these can lead to more changes. This process can repeat multiple times, until both the author and the reviewers approve of the change. We refer to each of these iterations as a *code snapshot*. Fig. 1 illustrates a code review with multiple code snapshots.

The many potential changes involved in the code review process create a challenge when empirically studying the effectiveness of individual analyses involved in contemporary code review in a real industrial setting. Because the code review typically involves at least two humans and tens if not hundreds of automated analyses, it can be difficult to attribute observations to individual analyses, in particular for more complex analyses like mutation testing, which reports mutants in code that require non-local changes to tests to be resolved.

Some analyses (e.g., ClangTidy) are usually limited to a single line and their findings can be fixed in the same line—pinpointing code that is almost always incorrect and should be fixed. If such an analysis surfaces an error in one snapshot but not a later one, it is likely that the presence of the analysis finding is associated with fixing that error. Reasoning about more complex analyses, like mutation testing, is not as straightforward: First, changes that address a mutant may be non-local (e.g., mutants are usually surfaced in application code, but changes that resolve them very often occur in the related test code). Second, multiple analyses can surface findings related to the same root cause on different lines, and resolving any one of the findings will resolve all of them. Third, as code changes it may be challenging to identify the exact new code location that needs to be mutated in order to determine whether a previously surfaced mutant has indeed been resolved.

In theory, any finding from an automated analysis might also have been identified by a human reviewer. However, automated analyses are much faster, cheaper, and never get tired. In our use case, it is highly unlikely that a human reviewer would inspect the code before automated analyses have finished.

B. Incremental Mutation Testing

Incremental mutation testing refers to surfacing undetected mutants in the context of a contemporary code review process [5]. It differs from traditional mutation analysis in two important ways. First, incremental mutation testing is scoped to a set of changed lines of code under review—as opposed to all lines of code in an existing code base. Second, incremental mutation testing is concerned with surfacing individual undetected mutants to developers—as opposed to computing an adequacy score (e.g., the mutant-detection ratio).

Fig. 1 illustrates how incremental mutation testing is integrated into the code review process. First, a developer begins their work by forking the latest state of the code repository into a feature branch. Then, they iterate on the code until they deem it ready for review. At any point, the developer can run

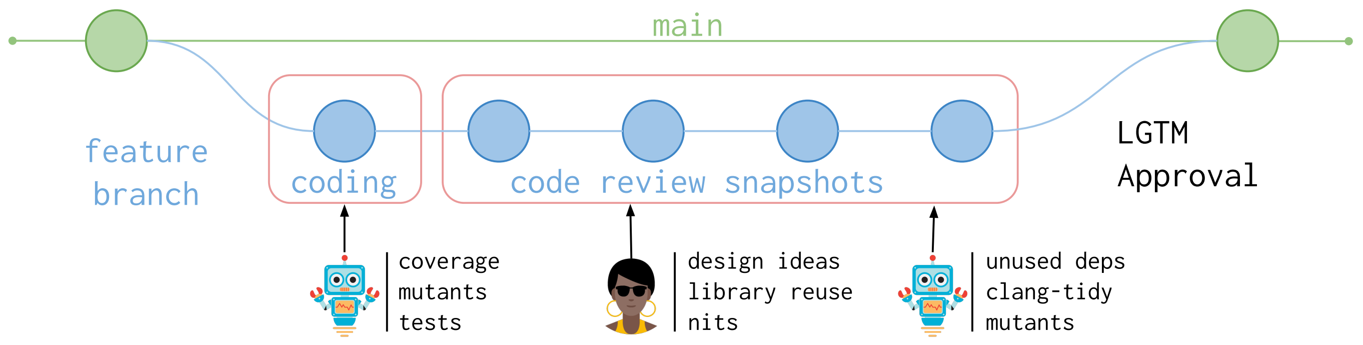


Fig. 1: Code evolves during code review over multiple code snapshots. During the initial coding phase, an author receives feedback from automated analyses. Once the code is sent for review, human reviewers contribute feedback, too.

configured automated analysis tools and observe code findings even before the code review starts. Finally, the developer sends the code for review by assigning reviewers. At this point, all configured automated analysis tools are run, and human reviewers are asked to review the code changes. Incremental mutation testing runs at the start of the review, usually before a human reviewer has had a chance to review the code, as well as on later snapshots, after the developer has changed the code in response to analysis results or human comments.

Code review can involve tens or hundreds of snapshots, significantly modifying earlier code changes. For example, code mutated in an initial snapshot can change, or disappear altogether, by the time a code change is approved and merged.

To study the effects of surfacing mutants in such a dynamic environment, it is critical to track the mutants as the mutated code gets modified, moved, or deleted during the code review. While a mutant-detection ratio provides a summary statistic over the detected mutants in any one snapshot, it provides no information about what happens to individual mutants and what actions a developer took to resolve them. As code evolves, more or fewer mutants may be generated between snapshots, and the mutation score may increase or decrease for that reason alone—without providing any insights into what happened to the previously surfaced mutants. Only by tracking individual mutants through the code review process can we arrive at conclusions about resolution strategies and outcomes.

C. Mutant Reaction

During code review, both the author and the reviewers of a change can react to a surfaced mutant. The user interface of the code review tool used in this paper provides two buttons:

- *Please fix (reviewers only)*: a reviewer believes that the mutant is productive —identifies an issue with the code or tests and should be resolved—and explicitly marks it as such. Clicking this button produces a human comment, with the reviewer's name and the text "Please fix". The author of the change must treat this comment like any other human comment. Specifically, the author must resolve all comments before merging the change. Author and reviewer can disagree and discuss the issue in multiple rounds of comments until they agree on the best way forward.
- *Not useful (reviewers or authors)*: a reviewer or the author believes that the mutant is unproductive —resolving the mutant is not necessary, undesirable, or may even lead to lower code or test quality.

The author and the reviewers are free to provide no reaction at all for surfaced mutants. The author can also resolve a mutant, without an explicit "please fix" reaction by a reviewer. For example, it is common to resolve automated findings before sending the merge request for code review, as indicated in Fig. 1. In this case, mutant resolution happened prior to assigning reviewers, and there is no explicit please-fix reaction.

D. Mutant Location

Given a code snapshot S^i, in which a mutant is reported on line L^i_n, and a follow-up code snapshot S^j $(j > i)$, there are five possible ways for the *mutated code location* to change:

- $L^i_n = L^j_n$: The mutated code is unmodified; an identical mutant can be generated in S^j on the same line.
- $L^i_n = L^j_m; n \neq m$: The mutated code is unmodified, but the code location has shifted; an identical mutant can be generated S^j on a different line.
- $L^i_n \sim L^j_n$: The mutated code is modified; a similar mutant can be generated in S^j on the same line.
- $L^i_n \sim L^j_m; n \neq m$: The mutated code is modified and the code location has shifted; a similar mutant can be generated S^j on a different line.
- $L^i_n \nsim L^j_m, \forall m$: The line containing the mutant was deleted or heavily modified. A similar mutant can no longer be generated in S^j (mutant invalidation).

Note that a snapshot index is unique w.r.t. a given merge request. We omit a merge-request index on S^i and L^i_n for clarity. Figs. 2 and 3 provide examples for a shift in code location as well as for identical and similar mutants.

E. Mutant Resolution

We say that a change between two code snapshots S^i and S^j $(j > i)$ *resolves* a mutant if (1) an undetected mutant exists in S^i, (2) an identical or similar mutant exists in S^j, and (3) a

```
11  int a = getB() + getC();  // mutated to -
12  return a;
13
```

```
11  setC();
12  int a = getB() + getC();  // mutated to -
13  return a;
```

Fig. 2: Identical mutant (`getB() + getC()` $\rightsquigarrow$ `getB() - getC()`) on line 11 in S^1 (left) and on line 12 in S^2 (right): the highlighted change (insertion of a new statement `setC()`) affects only non-mutated code, but shifts the mutated code location.

```
11  flush();
12  int a = b + c;  // mutated to -
13  return a;
```

```
11  flush();
12  int a = getB() + getC();  // mutated to -
13  return a;
```

Fig. 3: Similar mutant on line 12 in S^1 (left: `b + c` $\rightsquigarrow$ `b - c`) and in S^2 (right: `getB() + getC()` $\rightsquigarrow$ `getB() - getC()`): the highlighted change affects the mutated code, but the same mutation operator is still applicable on the same line.

test detects that mutant in S^j. A mutant in S^i and a mutant in S^j are *similar* if (1) they are generated by the same mutation operator and (2) the mutated AST nodes (modified expression or statement) are semantically related and similar. Section IV provides details about AST node relatedness and similarity.

F. Change After Mutant Intervention

Given a code snapshot S^i, which surfaced a mutant, we refer to the change between S^i and a subsequent snapshot S^j ($j > i$) as *change after mutant intervention*. Specifically, in this paper we consider the following four types of changes between the first code snapshot that surfaced a mutant and the code snapshot of the final merge:

1) *Code*: the change between the two code snapshots only affects non-test code.
2) *Tests*: the change between the two code snapshots only affects test code.
3) *Code+Tests*: the change between the two code snapshots affects both non-test and test code.
4) *Mutant invalidation*: the change between the two snapshots substantially modifies or deletes the mutated code location such that a similar mutant can no longer be generated.

About 5% of all mutants were invalidated between the snapshot that surfaced them and the snapshot of the final merge; we consistently exclude these when analyzing mutant resolution.

III. Data set

To answer our research questions, we produced a data set of 1,538 merge requests. This data set contains 2,806 unique, surfaced mutants, with different mutant reactions, as well as 5,127 unique, non-surfaced mutants, which were generated during code review but already detected by existing tests.

A. Merge Request Sampling

We sampled from a monolithic repository [17], containing more than 2 billion lines of code and thousands of projects written in various programming languages. The deployed mutation testing system supports 10 programming languages [18], and we sampled merge requests from the past six months at random, with no regard to programming language or project.

Given that the vast majority of generated mutants are either detected or receive no reaction, we used a stratified random sampling approach. Specifically, we randomly sampled 300 merge requests from each of the following three subpopulations, considering the difference between the initial submission and the final merge of each merge request:

1) *Please fix*: at least one mutant surfaced that received a please fix reaction.
2) *Not useful*: at least one mutant surfaced that received a not useful reaction.
3) *No reaction*: at least one mutant surfaced that received no reaction.

Additionally, we randomly sampled 900 merge requests from the following subpopulation:

4) *Not surfaced*: at least one mutant was generated and all generated mutants are detected, and hence none surfaced.

The goal of this stratified sampling approach was to increase the proportion of mutants that received a reaction without introducing further bias. Note that the first three subpopulations are not mutually exclusive. As a final step, we eliminated duplicate merge requests and those for which we were unable to determine mutant resolution for all surfaced mutants—resulting in a final data set of 1,538 merge requests.

B. Mutant Deduplication

Recall that a merge request usually evolves during code review. As a result mutation testing is run for multiple code snapshots. This means that an unresolved mutant may be surfaced multiple times and that a mutant may be surfaced for an arbitrary code snapshot. For example, a merge request may have no surfaced mutants initially, but changes introduced in a later code snapshot can lead to undetected mutants being surfaced. To aggregate the mutants in our data set that surfaced multiple times, we consider the first code snapshot that surfaces a particular mutant the point of intervention; later code snapshots for the same mutant are ignored. We consider the changes between the first code snapshot that surfaced a mutant and the final code snapshot of the merge request the change after intervention.

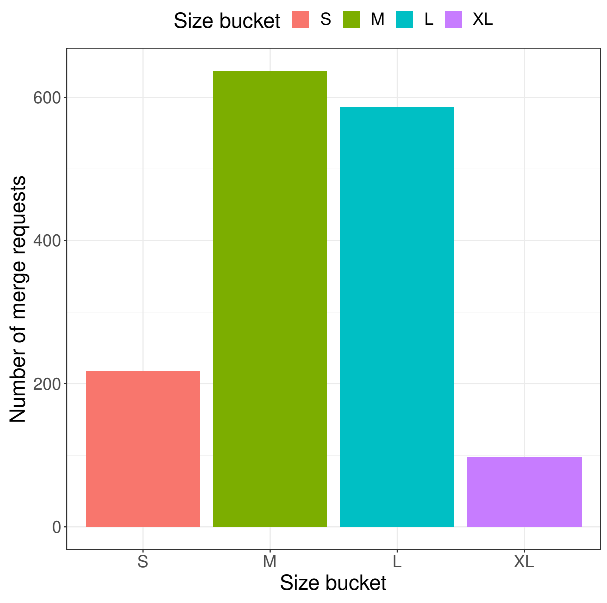


Fig. 4: Distribution of merge request sizes for all 1,538 merge requests. The size bucket of a merge request is determined by the number of changed lines in the initial code snapshot.

C. *Characteristics*

The final data set consists of 1,538 merge requests, with 2,806 surfaced and 5,127 non-surfaced mutants. Most of the surfaced mutants were generated for C++ (45%), followed by Java (19%) and Go (14%). The top-3 mutation operators that generated the surfaced mutants deleted statements/blocks (50% of mutants), mutated logical expressions (28% of mutants), and inserted unary operators (20% of mutants). Out of all 2,806 surfaced mutants, 319 received a please-fix reaction, 298 received a not-useful reaction, and 2,189 received no reaction.

Fig. 4 shows the size distribution of all merge requests, according to four size buckets. The size bucket of a merge request is determined by the number of changed lines (Δ_L) in the initial snapshot: S: $\Delta_L < 50$; M: $\Delta_L < 250$ L: $\Delta_L < 1000$; XL: $\Delta_L >= 1000$. We use a bucketing approach because the precise number of changed lines is irrelevant for our research, but we do expect the magnitude to confound some of our measures of interest (Section V). We determined these buckets empirically, considering the quantiles of the distribution of Δ_L.

IV. Tracking Mutants Across Code Snapshots

Determining mutant resolution requires tracking a mutant from the code snapshot in which it was surfaced to the final merged code snapshot. This is a non-trivial mapping problem, in particular if the source code sees substantial changes during code review. We devised a semi-automated approach to perform this mapping and manually validated the mapping for each surfaced mutant in our data set.

An initial exploration of a handful of merge requests revealed that mutated code changes surprisingly often. This observation motivated our first research question:

RQ1 *How often do code locations of surfaced mutants change as code evolves? Can these changes be accurately tracked?*

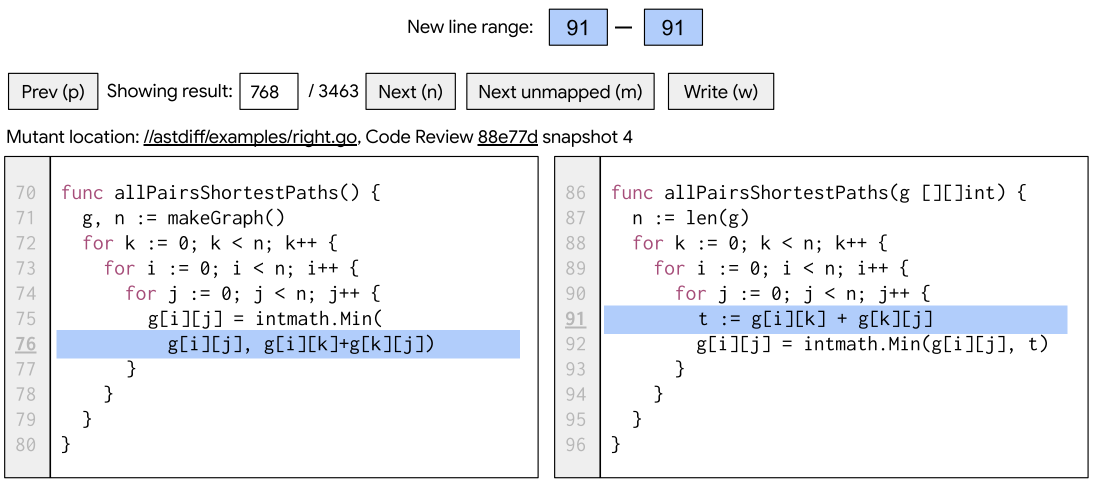
Being able to track mutants, or code findings more generally, is crucial for two reasons. First, when evaluating a mutant's effectiveness, it is necessary to understand what happens to the mutated source code as it evolves; specifically, determining whether a mutant was eventually resolved requires generating the same mutant in the final snapshot. Second, in order to consistently and accurately resurface the same mutant, if it remains undetected, requires repeatedly generating it for the same line of code, even if that line has moved or changed. The findings reported in Section V further reinforce the need for resurfacing mutants: a lack of resolution may go unnoticed if unresolved mutants are not resurfaced.

Traditionally, tree difference computation is a well-known research topic. However, optimal algorithms are prohibitively slow for any practical application that involves working with large trees, or even medium-sized trees for real-time applications such as interactive tools. Therefore, we use a tree-based diffing algorithm (a gumtree [19] variant) that uses heuristics tailored to the code base at hand for efficiency and scalability to large compilation units.

In our context, the result of applying this algorithm to a merge request is a mapping of abstract syntax tree (AST) nodes in one code snapshot to corresponding nodes in another snapshot. Given a mutated block of code in a code snapshot, this algorithm identifies syntactically and semantically equivalent or similar code in a later snapshot, even if the code changes to a significant degree. Put simply, it allows us to track where code moves as the code review progresses, and where mutants should be regenerated.

To measure the accuracy of this mutant-tracking approach and to ultimately answer RQ1, we developed a mapping tool, called MutantMapper, shown in Fig. 5. The tool provides a simple web interface that visualizes code diffs and highlights the mapping produced by the algorithm for each mutant. We developed the tool to perform a manual validation of the produced mappings. Using it, we were able to efficiently and semi-automatically map mutated code, even if it changed significantly during code review.

We manually analyzed all mutants in our data set, corrected the automatically computed mapping if needed, and classified the change between the two code snapshots as one of (1) same line, (2) different line, or (3) deleted line. If the mutated code was refactored beyond recognition, we classified it as deleted line. Note that we relied on MutantMapper for speed and convenience, not correctness. Overall, MutantMapper was highly accurate in mapping muated code locations; manual correction was required in less than 1% of all cases. The primary reason for an incorrect mapping was the existance of code clones. A mapping of mutated code locations can be established for other code bases, with or without MutantMapper as a convenience tool. We have no reason to believe that tree-based diffing would be substantially less effective on other code bases, but we leave a deeper investigation for future work.



New line range: 91 — 91

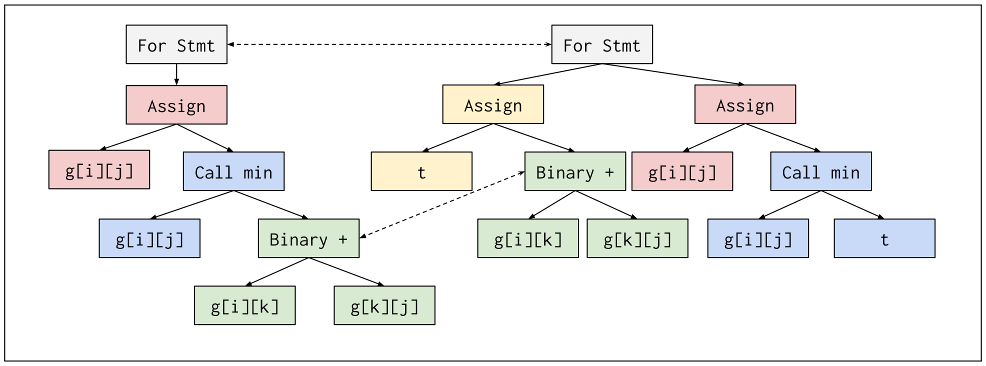
Prev (p) Showing result: 768 / 3463 Next (n) Next unmapped (m) Write (w)

Mutant location: //astdiff/examples/right.go, Code Review 88e77d snapshot 4

```
70 func allPairsShortestPaths() {
71   g, n := makeGraph()
72   for k := 0; k < n; k++ {
73     for i := 0; i < n; i++ {
74       for j := 0; j < n; j++ {
75         g[i][j] = intmath.Min(
76           g[i][j], g[i][k]+g[k][j])
77       }
78     }
79   }
80 }
```

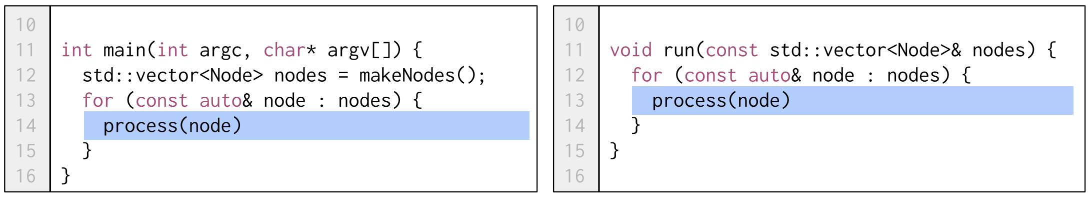
```
86 func allPairsShortestPaths(g [][]int) {
87   n := len(g)
88   for k := 0; k < n; k++ {
89     for i := 0; i < n; i++ {
90       for j := 0; j < n; j++ {
91         t := g[i][k] + g[k][j]
92         g[i][j] = intmath.Min(g[i][j], t)
93       }
94     }
95   }
96 }
```

(a) MUTANTMAPPER user interface visualizing the AST mapping below.



(b) Automated AST mapping example: mapping a binary operator node that is a child of a for loop node.

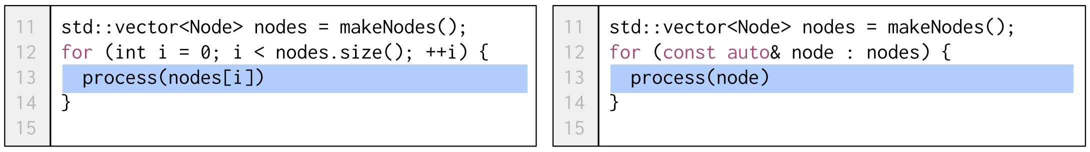
Fig. 5: Tracking an identical mutant (`g[i][k]+g[k][j]` $\rightsquigarrow$ `g[i][k]-g[k][j]`) from line 76 (left) to line 91 (right).



```

int main(int argc, char* argv[]) {
  std::vector<Node> nodes = makeNodes();
  for (const auto& node : nodes) {
    process(node)
  }
}
```

```

void run(const std::vector<Node>& nodes) {
  for (const auto& node : nodes) {
    process(node)
  }
}

```

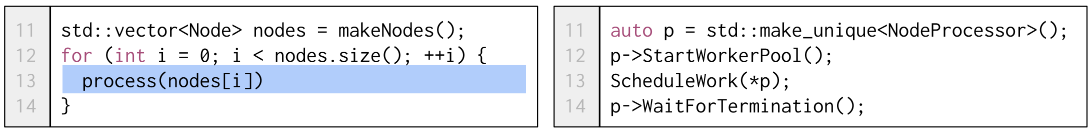
(a) Identical code: an identical mutant can be generated.



```
std::vector<Node> nodes = makeNodes();
for (int i = 0; i < nodes.size(); ++i) {
  process(nodes[i])
}

```

```
std::vector<Node> nodes = makeNodes();
for (const auto& node : nodes) {
  process(node)
}

```

(b) Similar code: a similar mutant can be generated.



```
std::vector<Node> nodes = makeNodes();
for (int i = 0; i < nodes.size(); ++i) {
  process(nodes[i])
}
```

```
auto p = std::make_unique<NodeProcessor>();
p->StartWorkerPool();
ScheduleWork(*p);
p->WaitForTermination();
```

(c) Refactored code: no similar mutant can be generated (mutant invalidation).

Fig. 6: Differences between code snapshots with varying degrees of change.

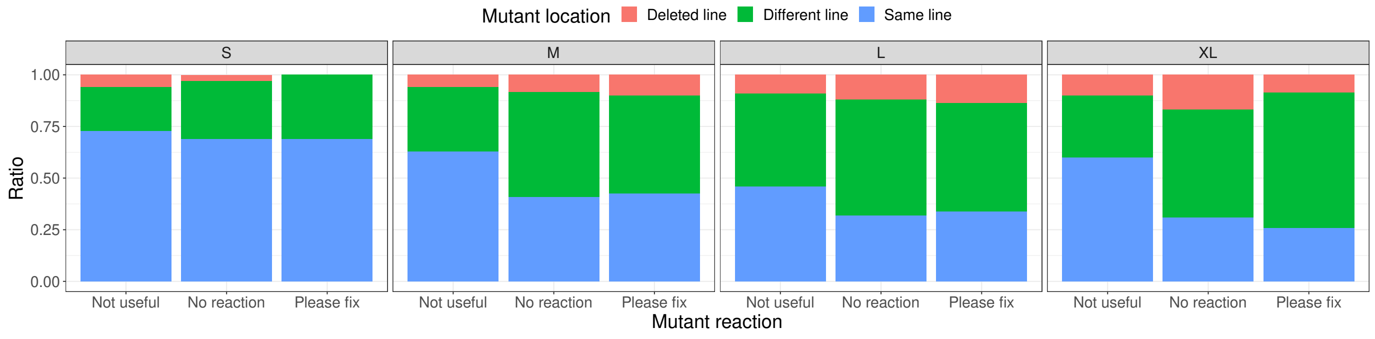


Fig. 7: Final mutant location, compared to where it was first surfaced, for all mutants broken down by merge request size.

Fig. 7 shows how often the location of the surfaced mutants changed during code review, mapping the mutated code location in the snapshot that surfaced a mutant to the corresponding location in the merged snapshot. The results are broken down by mutant reaction and lead to two important observations.

First, for all but the smallest merge requests, we observe a substantial location change. Indeed, the location of 64% of all mutants changes between the first snapshot that surfaces a mutant and the final snapshot of the corresponding merge request. These findings underscore the need for an accurate code mapping between snapshots that allows for accurate and consistent surfacing of mutants during code review.

Second, the extent to which the code changes during code review is noticeably smaller for not-useful mutants: not-useful mutants remain on the same line between 46% and 72% of the time whereas please-fix mutants remain on the same line between 26% and 69% of the time. This suggests that surfacing actionable mutants is associated with code changes, in addition to test changes.

A key takeaway is that tracking mutants is an important aspect of incremental mutation testing and that tree-based diffing is a viable solution. However, our particular tracking solution is only one point in the design space, and we leave a deeper exploration of possible design choices as well as efficiency and accuracy trade-offs for future work.

Summary (RQ1): The code location of 64% of all surfaced mutants changes during code review. Tree-based diffing can track such changes with an accuracy of over 99%.

V. Mutant Resolution

Having established the mapping (Section IV) for all surfaced mutants allowed us to automatically compute which of them were resolved during code review. This computation boils down to checking whether the mutant, regenerated in the mapped code in the merged snapshot, is detected—a mutant that is detected is resolved, otherwise it is unresolved.

The resolution data enables us to answer our second and third research questions:

RQ2 *How often are surfaced mutants resolved?*

RQ3 *Is mutant resolution associated with changes to both code and tests?*

For each research question, we are interested in contrasting the results by mutant reaction, specifically the difference between please-fix mutants and not-useful mutants. Additionally, we aim to answer each research question both quantitatively as well as qualitatively—thereby providing deeper insights for the quantified observations. For example, an important follow-up question to RQ2 is what the reasons for please-fix mutants remaining unresolved are when a merge request gets merged.

In the presence of multiple code findings and reviewer comments, it is generally impossible to determine what specific analysis prompted the code author to improve the code or add more tests in a post-hoc study. Many different code analyzers surface code findings in close proximity to one another and the causal path from a particular code finding to a related code improvement may be non-local. For example, in order to resolve a finding in one function a change may be required in an entirely different function. In other words, the attribution of code or test changes to a surfaced analysis finding is fuzzy at best. However, it is possible to establish whether the resolution of surfaced mutants is associated with code or test changes beyond the expected variance around the average changes for an arbitrary merge request.

A. *Measures of Interest*

a) Mutant reaction (Section II-C): Each surfaced mutant is labeled as one of: (1) please fix, (2) not useful, or (3) no reaction; non-surfaced mutants are labeled as (4) not surfaced.

b) Mutant resolution (Section II-E): Each surfaced mutant is labeled as either (1) resolved or (2) unresolved; each non-surfaced mutant is labeled as (3) "NA" since mutant resolution is not applicable in this case.

c) Change after mutant intervention (Section II-F): Each mutant is labeled as one of: (1) None—no change, (2) Code—changes only to non-test code, (3) Code+Tests—changes to both non-test and test code, or (4) Tests—changes only to test code. Since measuring lines of code is very sensitive to the corresponding programming language and can lead to comparability issues in a multi-language set up [8], we dichotomize code and test changes and assess the probability of change as opposed to the magnitude of that change.

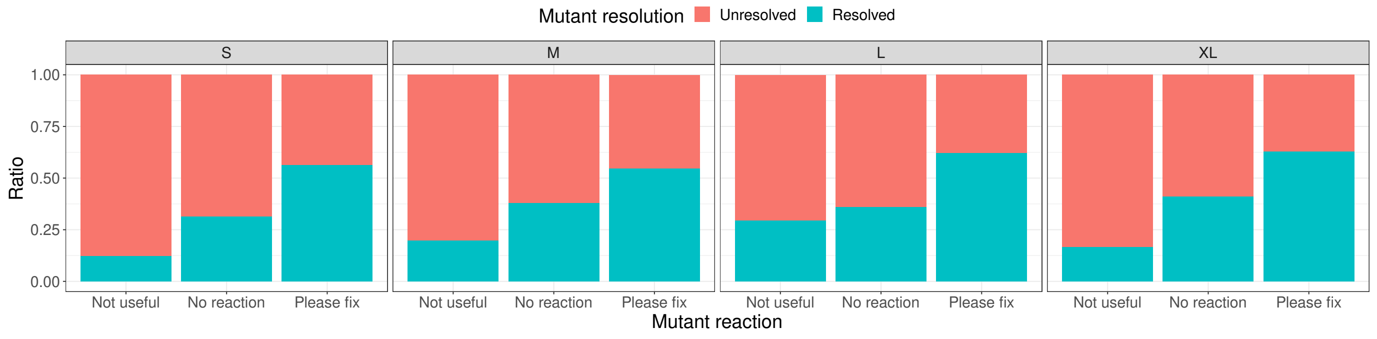


Fig. 8: Mutant resolution in the final snapshot, for all mutants broken down by mutant reaction and merge request size.

B. RQ2 How often are surfaced mutants resolved?

Fig. 8 shows the rate at which surfaced mutants are resolved, broken down by mutant reaction and merge request size. Overall, please-fix mutants are substantially more often resolved than mutants without an explicit reaction and more than twice as often resolved than not-useful mutants: not-useful mutants are resolved at a rate between 12% and 30%; no-reaction mutants are resolved at a rate between 31% and 41%; please-fix mutants are resolved at a rate between 56% and 63%. The overall resolution rate of please-fix mutants is about 60%. In contrast to mutant reaction, merge request size shows little influence on mutant resolution.

Summary (RQ2): Overall, 38% of all surfaced mutants and 60% of please-fix mutants are resolved. Please-fix mutants are resolved more than twice as often as not-useful mutants.

C. RQ3 Is mutant resolution associated with changes to both code and tests?

Fig. 9 shows how the change after intervention differs between resolved and unresolved mutants, broken down by mutant reaction. The plot also shows the distribution of changes for non-surfaced mutants to establish a baseline. (Note that these mutants were generated but already detected in the initial snapshot of a merge request, and hence the notion of mutant resolution is not applicable.) Fig. 9 shows a strong association between mutant resolution and changes to tests: the ratio of changes to only tests (purple area) is noticeably larger for resolved mutants. These findings are in line with prior work: they support the observation that surfacing undetected mutants motivates developers to write more tests [8]. The plot also shows that a large ratio of changes apply to both code and tests (green and blue areas).

To understand whether these observed changes represent significant deviations from the expectation, we fitted two logistic regression models and estimated the probability of change to code and tests, respectively, given the size bucket of a merge request, whether mutants surfaced, and the extent to which surfaced mutants are resolved in that merge request. Fig. 10 shows the estimated probability (and 95% confidence interval) that code or tests change for a merge request for which no mutant surfaced (red line) and for a merge request for which

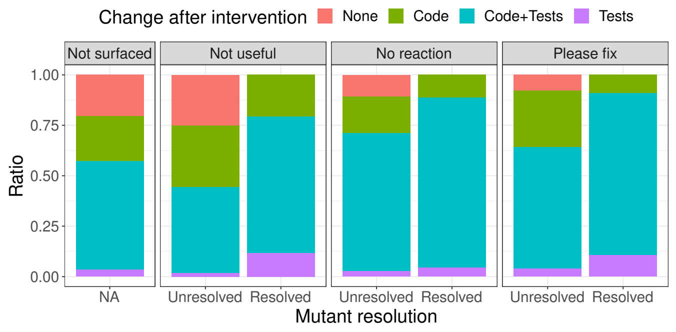


Fig. 9: Change after mutant intervention, for all mutants broken down by mutant reaction and resolution.

at least one mutant surfaced (blue line). Merge requests with non-surfaced mutants serve as a baseline and visual cue in the plot. Mutant resolution does not apply to these merge requests, and the ratio of resolved mutants is set to 0. (The red line is technically a single reference point, but we plot it as a horizontal line for visibility.)

For both changes to code and tests, merge request size is significantly associated with the probability of change ($p < 0.001$), with a strong effect. This observation is expected: larger merge requests tend to see more changes during code review. For test changes, the ratio of resolved mutants is significantly associated with the probability of change ($p < 0.001$), with a strong effect, independently of merge request size. The probability of change to tests is indistinguishable for merge requests with non-surfaced mutants (red line) and merge requests with surfaced but unresolved mutants (origin of the blue line with 0 resolved mutants). For code changes, the ratio of resolved mutants is weakly significantly associated with the probability of change ($p < 0.1$), with a non-negligible effect, independently of merge request size. Despite the baseline probability of changes to code being relatively high, mutant resolution still shows a measurable effect. Overall, the results suggest that mutant resolution is associated with changes to both code and tests.

Summary (RQ3): Merge requests with resolved mutants have a significantly higher probability of change to both code and tests, compared to merge requests with unresolved or non-surfaced mutants. The effect is stronger for tests than for code.

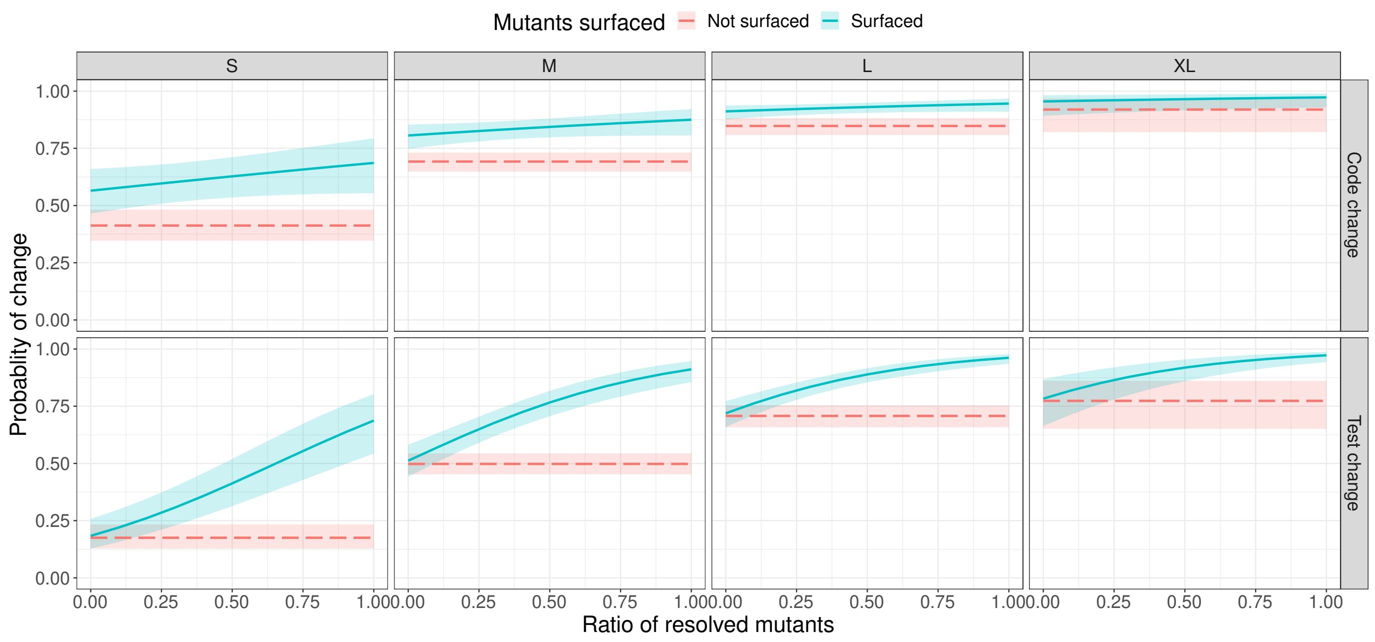


Fig. 10: Probability of change to code or tests as a function of the ratio of resolved mutants, broken down by merge request size. The shaded area gives the 95% confidence interval, and the dashed red line gives the probability of change for merge requests with not-surfaced mutants—merge requests for which all generated mutants were already detected, and hence none surfaced.

VI. Analysis of Resolved and Unresolved Mutants

To provide deeper insights into the findings presented in Section V, we manually analyzed merge requests with resolved and unresolved surfaced mutants, answering our final research question:

RQ4 *What actions do developers take to resolve mutants?*

For this analysis, we specifically focused on merge requests with please-fix mutants as these received a reviewer endorsement, and hence required action by the code author. Recall that during code review, the reviewer(s) can click the "please fix" button to signal that the code author should resolve a mutant.

We sampled 40% of merge requests for which (1) at least one mutant received a please-fix reaction, (2) no mutant received a not-useful reaction, and (3) all please-fix mutants were resolved. Additionally, we sampled 70% of merge requests for which (1) at least one mutant received a please-fix reaction, (2) no mutant received a not-useful reaction, and (3) all please-fix mutants were unresolved. We chose a higher sampling rate for merge requests with unresolved please-fix mutants as these are unexpected and might offer valuable insights. In total, we analyzed 51 merge requests with resolved please-fix mutants and 60 merge requests with unresolved please-fix mutants. For each of these merge requests, we analyzed the code review data and extracted the author-reviewer communication and mutant-resolution actions (i.e., test and code changes), if any.

A. Resolved Mutants

We followed a grounded-theory, open-coding approach. Specifically, one author performed an initial exploration, analyzing 20 resolved mutants and developing an initial code book. Then, all authors discussed the results, identified common themes, and refined the code book. Finally, one author used this code book to classify the remaining mutants.

TABLE I: Number of merge requests with resolved please-fix mutants per category.

Tests	Refactoring	Discussion	Inconclusive	Total
33	13	3	2	51

Four different categories of mutant resolution emerged, which are summarized in Table I: (1) in 33 cases, the code author added or modified a test to detect the mutant; (2) in 13 cases the code author refactored the code, to improve testability, and added tests; (3) in 3 cases, the mutants prompted a discussion between the author and the reviewer(s); (4) in 2 cases, the discussion and code/test changes were inconclusive.

1) Tests: Code authors resolved the majority of please-fix mutants by adding a new test. We did not observe any qualitative differences between the added tests and the ones that already existed in the code base.

2) Refactoring: If a mutant was surfaced in a code block that was not testable or hard to follow, the author first refactored the code and then added tests to detect the mutant. Often it was the case that the mutant could not be detected without the refactoring. A commonly observed theme was that the reviewer(s) asked an author to extract a complex piece of logic into a separate function, test it well, and use it at the call site.

3) Discussion: Mutants led to useful discussions, in particular for code-deletion mutations, where the reviewer(s) questioned whether a piece of code is redundant, or some additional data guards are required. This resulted in either the author explaining the reasoning, usually involving code externalities and project invariants that the reviewer might not be familiar with, resulting in better documentation. Another resolution path is the removal of clauses that are deemed redundant as the result of the discussion.

4) Inconclusive: Code review is an arena in which code authors and reviewers work together to produce the best possible merge request outcome. Sometimes, it is impossible to ascertain the causes of author actions. Mutants are shown in many code snapshots, and review comments are too. In two cases, after looking at the code review process, we were unable to establish with confidence why and how a mutant was resolved (i.e., whether the mutant elicited only test changes or also additional code changes).

B. Unresolved Mutants

Similar to resolved mutants, unresolved mutants in our data set involve some author-reviewer communication around the reviewers' request to address an endorsed mutant finding. Given that each mutant in this set is unresolved, such a mutant may have no attempted mutant resolution, or it may have an unsuccessful attempted mutant resolution. To analyze the code-review data for the unresolved please-fix mutants, we followed the same grounded-theory, open-coding approach as for resolved mutants.

We identified four different categories of reasons for a lack of (observed) mutant resolution, which are summarized in Table II: (1) in 27 cases, the code author pushed back on the request for resolving the mutant, mostly stating an argument for why adding a test for the mutant is undesirable; (2) in 16 cases, the author agreed with the mutant, but deferred the resolution for various reasons (3) in 3 cases, mutant resolution did occur but was unobservable due to the use of unusual testing approaches; (4) in 14 cases, mutant resolution was attempted but unsuccessful, a fact unknown to the code authors due to a limitation of the mutation testing system.

1) Pushback: Code authors questioned the importance of the test a surfaced mutant would have elicited, and its ability to detect actual bugs. As a result, the authors pushed back and did not resolve the surfaced mutant. We rely on the expertise of developers to make these decisions.

2) Deferred resolution: If the code under test is not designed with testing in mind, it might be difficult or impossible to write a test that detects the mutant, without a larger refactoring. In these cases, authors often left a `TODO` or created a tracking bug, leaving refactorings and additional tests to subsequent changes. This is common practice to separate the code review of a feature from additional quality improvements and ensure that these changes stand alone as individual commits in the version control system. For example, when the code change is either topical or should be possible to be rolled back in an emergency push, code authors push back on testing the mutant within the merge request under review. Additionally, larger refactorings and new tests, if they make up the bulk of the changes in the same merge request, could make the code review harder or the already selected reviewers less suitable.

TABLE II: Number of merge requests with unresolved please-fix mutants per category.

Pushback	Action deferred	Unusual testing	Infrastructure	Total
27	16	3	14	60

3) Unusual testing: Some mutants are surfaced despite being detected because of specialized tests that the mutation testing system is not aware of. For example, our mutation testing system supports JavaScript, Dart, and TypeScript, which are commonly used languages for frontend development. Some of the corresponding tests are large end-to-end tests or other user interface tests like screenshot tests. The mutation testing system does not evaluate such large tests, or it is not aware of them because there is no clear connection between, e.g., screenshot tests that assert on the equivalence or similarity between two browser screenshots and the code that produces the browser content. Since the mutation testing system only evaluates tests that directly exercise mutated lines of code, and no such connection exists, this class of tests are ignored. Thus, these mutants are resolved—a fact that can only be observed outside of the mutation testing system.

4) Infrastructure limitations: Code review is a complex process in which code findings present goals for developers to achieve: when developers take action to resolve a code finding, e.g., detect a mutant, they expect to see that mutant detected when the analysis reruns. Failure to provide that feedback produces a negative experience for the developer and is undesirable. Similarly, developers expect consistency and a clear set of goals during code review. For example, no additional mutants should be surfaced in subsequent snapshots of the code review, unless new code is added. Due to the probabilistic and incremental nature of our mutation testing system [5], additional mutants could in theory be generated and surfaced in subsequent snapshots. For example, if a restricted set of mutants (a subset of all possible mutants) surfaced in an initial snapshot and the code author adds tests to detect these mutants, the system could generate and surface additional mutants for the same code in a subsequent snapshot. This would result in an inconsistent user experience. To avoid that, the system only regenerates identical mutants in the same lines in which they initially surfaced. This means that detected mutants disappear and no new mutants are generated for previously unmutated code, thereby producing a consistent experience.

As code changes during code review, regenerating identical mutants on the same line of code is insufficient. Often, code authors add tests based on a mutant, but also update the code—moving the mutated line. In these cases, our mutation testing system did not regenerate the same mutant on the new line. As a result, code authors sometimes deemed a mutant resolved, even though the test written for it might have been inadequate.

Given that the code location of 64% of surfaced mutants changed (RQ1), the lack of an accurate mapping of mutants across snapshots accounted for 14 merge requests with unresolved please-fix mutants in our sample. While the code authors attempted mutant resolution for these merge requests, the mutation testing system did not resurface the undetected mutant(s) on subsequent snapshots. These observations prompted us to incorporate tree-based diffing for mutant tracking into our mutation testing system.

Summary (RQ4): The most common action for mutant resolution is additional testing, followed by code refactorings, and in-depth discussion about code relevance (in particular for statement-deletion mutants). The most common reasons for a lack of observed resolution are developers questioning the actionability of a mutant and deferred action (i.e., mutant resolution in a subsequent merge request).

VII. Related Work

There has been significant research interest in mutation testing in recent years. However, relatively few publications provide data or insights into human interactions with mutation testing. More common is the use of simulations on historical or synthetic datasets (e.g., [2], [4], [20]).

A. Industrial Applications of Mutation Testing

Prior works by Google [8], [18] and Facebook [9] describe deployments of mutation testing at industrial scale, together with associated challenges and technical solutions. They also provide some anecdotal data on user satisfaction, but mostly in the form of supporting evidence for the technical contributions.

A case study by Ahmed et al. [21] reports on an application of mutation testing on a Linux Kernel module. The study analyzed 3169 mutants, out of which 380 were live. The study examined the computational costs of mutation analysis and ways in which it could be reduced. It concludes that "mutation testing can and should be more extensively used in practice".

Delgado-Pérez et al. [22] report on an application of mutation testing in the nuclear industry. The subjects of the study were firmware modules, with 15 functions ranging from 10 to 63 lines of code. The authors examined 2509 mutants, out of which 154 were live. The aim of the study was to determine computational costs and human effort caused by equivalent mutants when extending a coverage-adequate test suite to a mutation-adequate one. The main insight is that mutation testing can indeed improve fault detection compared to structural-coverage-guided testing.

In contrast to these prior studies, our work specifically probes into the interactions between developers and surfaced mutants.

B. Gamification of Mutation Testing

Rojas and Fraser [23] and Rojas et al. [24] present a mutation testing game played between humans. The focus of this research was to examine if gamification can be used to produce high quality mutants and to improve test suites. Rojas et al.'s research is perhaps the closest to the study presented in this paper, but it uses different interventions to modify human behavior by introducing goals and rewards; their data could not be used to answer the research questions presented in this paper.

C. Actionability of Surfaced Mutants

The notion that some mutants are more valuable and/or harder to detect than others has also received attention in the research community. Common themes in prior work were attempts to eliminate redundancy among mutants or identify a sufficient subset of mutants (e.g., [25]–[29]) and to rank mutants by utility (e.g., [20], [30], [31]).

In contrast to these prior studies, our work specifically ties the actionability of mutants to developer feedback and observable actions taken to resolve surfaced mutants.

VIII. Conclusions

This paper reports on an empirical study that analyzes how developers resolve mutants surfaced during code review, using a dataset of 1,538 merge requests with 7,933 mutants. The paper's key results and implications are as follows.

First, the code location of 64% of all surfaced mutants changes during code review. This large percentage demonstrates the need for advanced code mapping techniques that can track mutants across snapshots during the code review process. An accurate mapping is important for both researchers, who wish to evaluate the effects of mutation testing, and engineers who wish to develop effective mutation testing systems. Our results suggest that it is possible to adapt tree-based diffing to efficiently track mutants across snapshots, with an accuracy of 99%. We specifically adapted the diffing algorithm used in this paper to our codebase. While this codebase is very large, it remains an open question to what extent tree-based diffing is effective for evolving merge requests in other codebases.

Second, developers resolve 38% of all surfaced mutants. The resolution rate is 60% for mutants endorsed by reviewers during code review, and developers resolve endorsed mutants more than twice as often as mutants they deem not useful. Additionally, merge requests with resolved surfaced mutants have a significantly higher probability of change to both code and tests, compared to merge requests with unresolved or non-surfaced mutants. Overall, our results suggest that mutation testing offers benefits beyond improved testing. While the most common action for mutant resolution was adding additional tests, we additionally observed code refactorings and in-depth discussions prompted by surfaced mutants. We also observed that the most common reason for a lack of resolution is developers questioning the value of certain mutants, considering them not actionable or the required effort to resolve them not worthwhile. Given these findings, we plan to research techniques for increasing the ratio of mutants developers endorse while minimizing the ratio of mutants they consider not useful.

References

[1] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2014, pp. 189–200.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005, pp. 402–411.

[3] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, Nov. 2014, pp. 654–665.

[4] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, “Revisiting the relationship between fault detection, test adequacy criteria, and test set size,” in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, September 21–25 2020, pp. 237–249.

[5] G. Petrović and M. Ivanković, “State of mutation testing at Google,” in *Proceedings of the International Conference on Software Engineering—Software Engineering in Practice (ICSE SEIP)*, May 2018.

[6] G. Petrović, M. Ivanković, B. Kurtz, P. Ammann, and R. Just, “An industrial application of mutation testing: Lessons, challenges, and research directions,” in *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, Apr. 2018, pp. 47–53.

[7] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” *Commun. ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3188720

[8] G. Petrović, M. Ivanković, G. Fraser, and R. Just, “Does mutation testing improve testing practices?” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021, pp. 910–921.

[9] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, “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)*, 2021, pp. 268–277.

[10] H. Coles, “Mutation testing — a practitioner’s perspective,” https://sites.google.com/site/mutation2017/keynote.

[11] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721.

[12] P. C. Rigby and C. Bird, “Convergent contemporary software peer review practices,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.

[13] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, “Development and deployment at facebook,” *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

[14] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, “Modern code review: A case study at google,” in *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[15] T. C. Team, “Clang-Tidy,” https://clang.llvm.org/extra/clang-tidy/.

[16] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, “Building useful program analysis tools using an extensible java compiler,” in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 14–23.

[17] R. Potvin and J. Levenberg, “Why Google stores billions of lines of code in a single repository,” *Communications of the ACM*, vol. 59, pp. 78–87, Jul. 2016.

[18] G. Petrović, M. Ivanković, G. Fraser, and R. Just, “Practical mutation testing at scale: A view from google,” *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 10, pp. 3900–3912, 2021.

[19] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[20] S. J. Kaufman, R. Featherman, J. Alvin, B. Kurtz, P. Ammann, and R. Just, “Prioritizing mutants to guide mutation testing,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 25–27 2022.

[21] I. Ahmed, R. Gopinath, C. Jensen, A. Groce, and P. E. McKenney, “Applying mutation analysis on kernel test suites: An experience report,” in *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, March 2017, pp. 110–115.

[22] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. Clark, and I. Medina-Bulo, “Evaluation of mutation testing in a nuclear industry case study,” *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1406–1419, 2018.

[23] J. M. Rojas and G. Fraser, “Code defenders: a mutation testing game,” in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2016, pp. 162–167.

[24] J. M. Rojas, T. D. White, B. S. Clegg, and G. Fraser, “Code defenders: crowdsourcing effective tests and subtle mutants with a mutation testing game,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 677–688.

[25] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, “Measuring effectiveness of mutant sets,” in *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2016, pp. 132–141.

[26] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, “Mutation subsumption graphs,” in *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, Cleveland, Ohio, USA, March 2014, pp. 176–185.

[27] P. Ammann, M. E. Delamaro, and J. Offutt, “Establishing theoretical minimal sets of mutants,” in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, March 2014, pp. 21–31.

[28] X. Yao, M. Harman, and Y. Jia, “A study of equivalent and stubborn mutation operators using human analysis of equivalence,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2014, pp. 919–930.

[29] M. Kintis, M. Papadakis, and N. Malevris, “Evaluating mutation testing alternatives: A collateral experiment,” in *17th Asia Pacific Software Engineering Conference (APSEC2010)*, Sydney, Australia, November 2010.

[30] A. Namin, X. Xue, O. Rosas, and P. Sharma, “Muranker: A mutant ranking tool,” *Software Testing, Verification and Reliability (JSTVR)*, vol. 25, no. 5-7, pp. 572–604, 2015.

[31] R. Just, R. J. Kurtz, and P. Ammann, “Inferring mutant utility from program context,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2017, pp. 284–294.