Practical Mutation Testing at Scale: A view from Google

Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just

Abstract—Mutation analysis assesses a test suite's adequacy by measuring its ability to detect small artificial faults, systematically seeded into the tested program. Mutation analysis is considered one of the strongest test-adequacy criteria. Mutation testing builds on top of mutation analysis and is a testing technique that uses mutants as test goals to create or improve a test suite. Mutation testing has long been considered intractable because the sheer number of mutants that can be created represents an insurmountable problem–both in terms of human and computational effort. This has hindered the adoption of mutation testing as an industry standard. For example, Google has a codebase of two billion lines of code and more than 150,000,000 tests are executed on a daily basis. The traditional approach to mutation testing does not scale to such an environment; even existing solutions to speed up mutation analysis are insufficient to make it computationally feasible at such a scale. To address these challenges, this paper presents a scalable approach to mutation testing based on the following main ideas: (1) mutation testing is done incrementally, mutating only *changed code* during code review, rather than the entire code base; (2) mutants are filtered, removing mutants that are likely to be irrelevant to developers, and limiting the number of mutants per line and per code review process; (3) mutants are selected based on the historical performance of mutation operators, further eliminating irrelevant mutants and improving mutant quality. This paper empirically validates the proposed approach by analyzing its effectiveness in a code-review-based setting, used by more than 24,000 developers on more than 1,000 projects. The results show that the proposed approach produces orders of magnitude fewer mutants and that context-based mutant filtering and selection improve mutant quality and actionability. Overall, the proposed approach represents a mutation testing framework that seamlessly integrates into the software development workflow and is applicable to industrial settings of any size.

Index Terms—Mutation testing, code coverage, test efficacy

1 Introduction

Software testing is the predominant technique for ensuring software quality, and various approaches exist for assessing test suite efficacy (i.e., a test suite's ability to detect software defects). A common approach is code coverage, which is widely used at Google [1] and measures the degree to which a test suite exercises a program. Code coverage is intuitive, cheap to compute, and well supported by commercial-grade tools. However, code coverage alone is insufficient and may give a false sense of efficacy, in particular if program statements are covered but their expected outcome is not asserted upon [2], [3]. An alternative approach that addresses this limitation is *mutation analysis*, which systematically seeds artificial faults, called mutants, into a program and measures a test suite's ability to detect them [4]. Mutation analysis addresses is widely considered the best approach for evaluating test suite efficacy [5], [6], [7]. *Mutation testing* is an iterative testing approach that builds on top of mutation analysis and uses undetected mutants as concrete test goals to guide the testing process.

• Goran Petrović and Marko Ivanković are with Google LLC, 8002 Zürich, Switzerland. E-mail: {goranpetrovic, markoi}@google.com.
• Gordon Fraser is with the University of Passau, 94032 Passau, Germany. E-mail: gordon.fraser@uni-passau.de.
• René Just is with the University of Washington, Seattle, WA 98105 USA. E-mail: rjust@cs.washington.edu.

Manuscript received 13 June 2020; revised 27 July 2021; accepted 14 August 2021. Date of publication 27 August 2021; date of current version 17 October 2022. (Corresponding author: Goran Petrović.) Recommended for acceptance by W. Visser. Digital Object Identifier no. 10.1109/TSE.2021.3107634

As a concrete example, consider the following fully covered, yet weakly tested, function `view`:

```
public Buffer view() {
  Buffer buf = new Buffer();
  // mutation: delete this line
  buf.append(this.internal_buf);
  return buf;
}

public void testView() {
  Buffer b = new Buffer("internal buffer");
  assertNotNull(b.view());
}
```

The test exercises the function, but does not assert upon its effects on the returned buffer. In this case, mutation analysis outperforms code coverage: even though the line that appends some content to `buf` is covered, a developer is not informed about the fact that no test checks for its effects. The statement-deletion mutation highlighted in the code example explicitly points out this testing weakness: the test does not fail when inserting this artificial defect.

Google always strives to improve test quality, and thus decided to implement and deploy mutation testing to evaluate its efficacy. The scale of Google's code base with approximately 2 billion lines of code, however, rendered the traditional approach to mutation testing infeasible: more than 150,000,000 test executions per day are gatekeepers for 40,000 change submissions to this code base, ensuring that 14,000 continuous integrations remain healthy on a daily basis [8], [9]. First, systematically

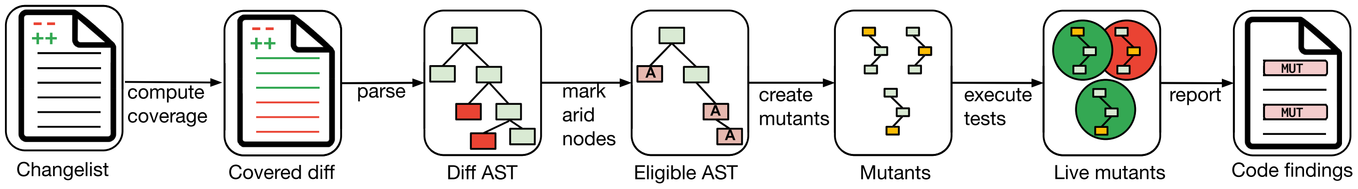


Fig. 1. The Mutation Testing Service. For a given changelist, line coverage is computed and the code is parsed into an AST. For AST nodes spanning covered lines, arid nodes are marked, using the arid node detection heuristics, and only non-arid (eligible) nodes are mutated. The generated mutants are tested, and surviving mutants are reported as code findings.

mutating the entire code base, or even individual projects, creates a substantial number of mutants, each potentially requiring the execution of many tests. Second, neither the traditionally computed mutant-detection ratio, which quantifies test suite efficacy, nor simply showing all mutants that have evaded detection to a developer would be actionable. Given that resolving a single mutant takes several minutes [10], [11], the required developer effort for resolving all undetected mutants would be prohibitively expensive, even at a small scale.

To make matters worse, even when applying sampling techniques to substantially reduce the number of mutants, developers at Google initially classified 85% of reported mutants as unproductive. An *unproductive* mutant is either trivially equivalent to the original program or it is detectable, but adding a test for it would not improve the test suite [11]. For example, mutating the initial capacity of a Java collection (e.g., `new ArrayList(64)` $\mapsto$ `new ArrayList(16)`) creates an unproductive mutant. While it is possible to write a test that asserts on the collection capacity or expected memory allocations, it is unproductive to do so. In fact, it is conceivable that these tests, if written and added, would even have a negative impact because their change-detector nature (specifically testing the current implementation rather than the specification) violates testing best practices and causes brittle tests and false alarms.

Faced with the two major challenges in deploying mutation testing—the computational costs of mutation analysis and the fact that *most mutants are unproductive*—we have developed a mutation testing approach that is scalable and usable, based on three central ideas:

1) Our approach *performs mutation testing on code changes*: it considers only *changed lines* of code and reports mutants during code review (Section 2, based on our prior work [12]). This greatly reduces the number of lines in which mutants are created and matches a developer's unit of work for which additional tests are desirable.
2) Our approach uses *transitive mutant suppression*: it uses heuristics based on developer feedback (Section 3, based on our prior work [12]). The feedback of more than 20,000 developers on thousands of mutants over six years enabled us to develop heuristics for mutant suppression that reduce the ratio of unproductive mutants from 85% to 11%.
3) Our approach uses *probabilistic, targeted mutant selection*: it reports a restricted number of mutants based on historical mutant performance, further avoiding unproductive mutants (Section 4).

Our evaluation of the proposed approach involved 760,000 code changes and 2 million mutants reported during code review, out of a total of almost 17 million generated mutants (Section 5). The results show that our approach makes mutation testing feasible and actionable—even for industry-scale software development environments.

2 Mutation Testing at Google

Mutation testing at Google faces challenges of scale, both in terms of computation time as well as integration into the developer workflow. Even though existing work on selective mutation and other optimizations [13] can substantially reduce the number of mutants that need to be analyzed, it remains prohibitively expensive to compute the mutant-detection ratio for Google's entire codebase due to its size. It would be even more expensive to keep re-computing the mutant-detection ratio, e.g., on a daily or weekly basis, and it is infeasible to compute it after each commit. In addition to the costs of computing that ratio, we were unable to find a good way to report it to the developers in an actionable way: it is neither concrete nor actionable, and it does not guide testing. Reporting individual mutants at scale to developers is also challenging, in particular due to unproductive mutants.

Addressing the challenges of scale and unproductive mutants, we designed and implemented a mutation testing approach that differs from the traditional approach, described in the literature [14]. For scalability, we designed and implemented *diff-based mutation testing*, which only generates and evaluates mutants for covered, changed lines; for productivity, we designed and implemented an approach for *mutant suppression and probabilistic mutant selection*.

Mutation testing at Google starts when a developer sends a code change for code review. The mutation testing process consists of four high-level steps: code coverage analysis (Section 2.1), mutant generation (Section 2.2), mutation analysis (Section 2.3), and reporting surviving mutants in the code review process (Section 2.4).

Fig. 1 details the Mutation Testing Service. (1) It starts with a changelist submitted for code review. (2) Once code-coverage metadata is available, it determines the set of lines that are covered, and added or modified in the changelist. (3) It then constructs an AST of each affected file and visits each covered node. (4) It then labels arid nodes (nodes that if mutated create unproductive mutants), based on the heuristics accumulated using developer feedback about mutant productivity over the years. Arid node labeling happens before mutants are generated, and hence mutants in arid nodes are never generated in the first place. (5) Mutagenesis

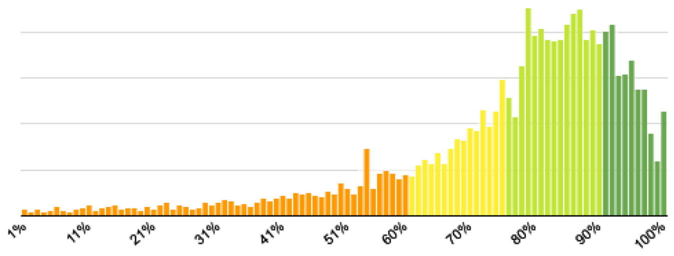


Fig. 2. Distribution of project line coverage.

then generates mutants for eligible nodes (i.e., each node that is not arid and that is covered by at least one test). (6) The Mutation Testing Service then evaluates the mutants against the existing tests, and (7) reports a subset of surviving mutants as code findings in the code review.

2.1 Prerequisites: Changelists and Coverage

A *changelist* is an atomic update to the version control system, and it consists of a list of files, the operations to be performed on these files, and possibly the file contents to be modified or added, along with metadata like change description, author, etc.

Once a developer sends a changelist to peer developers for code review, various static and dynamic analyses are run for that changelist and *findings* are reported to the developer and the reviewers. Line coverage is one such analysis: during code review, *overall* and *delta* code coverage is reported to the developers [1]. Overall code coverage is the ratio of the number of lines covered by tests in the file to the total number of instrumented lines in the file. The number of instrumented lines is usually smaller than the total number of lines, since artifacts like comments or pure whitespace lines are excluded. Delta coverage is the ratio of number of covered added or modified lines to the total number of added or modified lines in the changelist. Fig. 2 shows the line-coverage distribution per project, indicating that line coverage of most projects is satisfactory.

Code coverage is a prerequisite for running mutation analysis due to the high cost of generating and evaluating mutants in uncovered lines, all of which would inevitably survive because the code is not tested. Once line-level coverage is available for a changelist, mutagenesis is triggered.

Google uses Bazel as its build system [15]. Build targets explicitly list their sources and dependencies, and correspond to an arbitrary number of test targets, each of which can involve multiple tests. Tests are executed in parallel. Using the explicit dependency and source listing, the code-coverage analysis provides information about which test target covers which lines in the source code, thereby linking lines of code to a set of tests covering them. Line-level coverage is used to determine the set of tests that need to be run in an attempt to kill a mutant. This approach is also implemented in other mutation testing tools, including PIT [16] and Major [17], [18].

2.2 Mutagenesis

The mutagenesis service receives a request to generate *point mutations*, i.e., mutations that produce a mutant which differs from the original in one AST node on the requested line. For each supported programming language, a special mutagenesis service capable of navigating the AST of a compilation unit in that language accepts point mutation requests and replies with potential mutants. The mutation operators are implemented as AST visitors, an approach also taken by other mutation tools (e.g., [19]). For each point mutation request, i.e., a $(file, line)$ tuple, a mutation operator is selected and a mutant is generated in that line if that mutation operator is applicable to it. If no mutant is generated by the mutation operator, another operator is selected and so on until either a mutant is generated or all mutation operators have been tried and no mutant could be generated. There are two mutation operator selection strategies, *random* and *targeted*, detailed in Section 4.

The Mutation Testing Service generates at most one mutant per line, for scalability reasons and based on the insight that the vast majority of mutants for a given line share the same fate—either all or none of them survive the analysis [20]. This means that if a mutant generated for a given line does not survive the mutation analysis, no additional mutants are generated for that line.

The Mutation Testing Service implements mutagenesis for 10 programming languages: C++, Java, Go, Python, TypeScript, JavaScript, Dart, SQL, Common Lisp, and Kotlin. For each language, the service implements five mutation operators: AOR (Arithmetic operator replacement), LCR (Logical connector replacement), ROR (Relational operator replacement), UOI (Unary operator insertion), and SBR (Statement block removal). These mutation operators were originally introduced for Mothra [21], and Table 1 gives an example for each. In Python, unary increment and decrement are replaced by a binary operator to achieve the same effect due to the language design. In our experience, the ABS (Absolute value insertion) mutation operator predominantly created unproductive mutants, mostly because it acted on time-and-count related expressions, which are positive and nonsensical if negated. Therefore, the Mutation Testing Service does not use the ABS operator. Note that our observations may not hold in general and may be a function of the style and features of our codebase.

TABLE 1
Implemented Mutation Operators

Operator	Example
AOR	$\texttt{a + b} \rightarrow \{\texttt{a}, \texttt{b}, \texttt{a} - \texttt{b}, \texttt{a} * \texttt{b}, \texttt{a}/\texttt{b}, \texttt{a\%b}\}$
LCR	$\texttt{a \&\& b} \rightarrow \{\texttt{a}, \texttt{b}, \texttt{a}\|\|\texttt{b}, \texttt{true}, \texttt{false}\}$
ROR	$\texttt{a > b} \rightarrow \{\texttt{a} < \texttt{b}, \texttt{a} <= \texttt{b}, \texttt{a} >= \texttt{b}, \texttt{true}, \texttt{false}\}$
UOI	$\texttt{a} \rightarrow \{\texttt{a}++, \texttt{a}--\}$ (also $\texttt{a} \rightarrow \texttt{!a}$, etc.)
SBR	$\texttt{stmt} \rightarrow \emptyset$

2.3 Mutation Analysis

Once mutagenesis has generated a set of mutants for a changelist, a temporary state of the version control system is prepared for each of them, based on the original changelist, and then tests are executed in parallel for all those states. This allows for an efficient interaction and caching between our version control system and build system, and evaluates mutants in the fastest possible manner.

Once the mutation analysis results are available, the Mutation Testing Service selects and reports mutants from the set of surviving mutants. We limit the number of reported mutants to at most 7 times the number of total files in a changelist. This ensures that the cognitive overhead of understanding all reported mutants is not too high, which might otherwise cause developers to stop using mutation testing. We empirically determined 7 to be an appropriate trade-off between test efficacy and cognitive load by collecting data over the years of running the system. Finally, the service reports selected surviving mutants in the code review UI to the author and the reviewers. Note that for consistency, the Mutation Testing Service selects and reports mutants in the same line(s) as before if an author adds additional tests or otherwise updates the changelist, which triggers a re-execution of the service.

2.4 Reporting Mutants in the Code Review Process

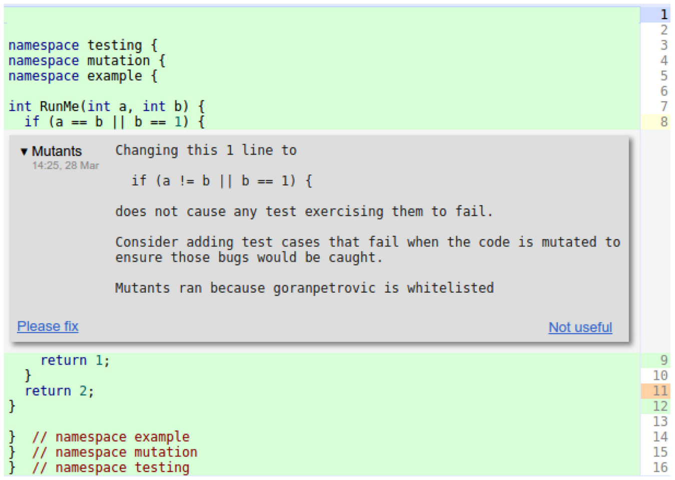
Most changes to Google's codebase, except for a limited number of fully automated changes, are reviewed by developers before they are merged into the source tree. Potvin and Levenberg [9] provide a comprehensive overview of Google's development ecosystem. Reviewers can leave comments on the changed code that must be resolved by the author. A special type of comment generated by an automated analyzer is known as a *finding*. Unlike human-generated comments, findings do not need to be resolved by the author before submission, unless a human reviewer marks them as mandatory. Many analyzers are run automatically when a changelist is sent for review: linters, formatters, static code and build dependency analyzers, etc. The majority of analyzers are based on the Tricorder code analysis platform [22].

The Mutation Testing Service reports selected mutants to developers during the code review process, which maximizes the chances that these will be considered by the developers. The number of comments displayed during code review can be large, so it is important that all tools produce actionable findings that can be used immediately by the developers. Reporting non-actionable findings during code review has a negative impact on the author and the reviewers. If a finding (e.g., a surviving mutant) is not perceived as useful, developers can report that with a single click on the finding. If any of the reviewers consider a finding to be important, they can indicate that to the changelist author with a single click. Fig. 3 shows an example mutant displayed in Critique, Google's Code Review system[23], including the "Please Fix" and "Not useful" links in the bottom corners. This feedback is accessible to the owner of the system that created the findings, so quality metrics can be tracked, and non-actionable findings triaged and ideally prevented in the future.

To be of any use to the author and the reviewers, code findings need to be actionable and reported quickly, before the review is complete. To that end, the Mutation Testing Service performs mutant suppression (Section 3), and it probabilistically selects mutants based on their historical mutation operator performance (Section 4).

3 Suppressing Unproductive Mutants

Some parts of the code are less interesting than others. Reporting live mutants in uninteresting statements (e.g., logging statements for debugging purposes) has a negative



```
namespace testing {
namespace mutation {
namespace example {

int RunMe(int a, int b) {
  if (a == b || b == 1) {
```

```
Changing this 1 line to

  if (a != b || b == 1) {

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to
ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted
```

```
    return 1;
  }
  return 2;
}

}  // namespace example
}  // namespace mutation
}  // namespace testing
```

Fig. 3. Mutant reported in the code review tool.

impact on cognitive load and time spent analyzing mutants. Because developers do not perceive adding tests to kill mutants in uninteresting code as improving the overall efficacy of the test suite, such mutants tend to survive and be flagged as unproductive.

This section proposes an approach for suppressing unproductive mutants, based on a set of heuristics for detecting arid (i.e., uninteresting) AST nodes. There is a trade-off between correctness and usability of the results; a heuristic may prevent a mutation in very few non-arid nodes as a side-effect of suppressing mutations in many arid nodes. We argue that this is a good trade-off because the number of possible mutants is orders of magnitude larger than what the mutation service could reasonably report to the developers within the existing developer tools. Moreover, preventing non-actionable findings is more important than reporting all actionable findings.

3.1 Detecting Arid Nodes

In order to prevent the generation of unproductive mutants, the Mutation Testing Service identifies arid nodes in the AST, which are related to uninteresting statements. Examples of arid nodes include calls to memory-reserving functions like `std::vector::reserve` and writing to `stdout`; these are typically not tested by unit tests.

Mutation operators create mutants based on the AST of a program. The AST contains nodes, which are statements, expressions or declarations, and their child-parent relationships reflect their connections in the source code [24]. Most compilers differentiate between simple and compound AST nodes. Simple nodes have no body; for example, a function-call expression provides a function name and arguments, but has no body. Compound nodes have at least one body; for example, a `for` loop might have one body, while an `if` statement might have two—the `then` and `else` branches.

Our heuristics-based approach for labeling nodes as arid is two-fold:

$$arid(N) = \begin{cases} expert(N) & if\ simple(N) \\ 1\ if \bigwedge(arid(c)) = 1, \forall c \in N & otherwise \end{cases}. \tag{1}$$

Here, $N \in T$ is a node in the abstract syntax tree T of a program, *simple* is a boolean function determining whether a node is a simple or compound node (compound nodes contain their children nodes c), and *expert* is a partial boolean function mapping a subset of simple nodes in T to the property of being arid. The first part of Equation (1) operates on simple nodes, using the expert function, which encodes knowledge that is manually curated for each programming language and adjusted over time. The second part operates on compound nodes and is defined recursively. A compound node is arid iff *all* of its children nodes are arid.

The *expert* function flags simple nodes as arid and is based on developer feedback on reported "Not useful" mutants. This is a manual process: if we determine that a certain mutant is indeed unproductive and that an entire class of such mutants should not be created, a rule is added to the *expert* function. This is a key component of the Mutation Testing Service—without it, users would become frustrated with non-actionable findings and opt out of the system altogether. Targeted mutation and careful reporting of mutants have been crucial for the adoption of mutation testing at Google. So far, we have accumulated more than one hundred rules for arid node detection.

3.2 Expert Heuristic Categories

The *expert* function consists of various rules, some of which are mutation-operator-specific, and some of which are universal. We distinguish between heuristics that prevent the generation of uncompilable versus compilable yet unproductive mutants. Most heuristics deal with the latter category, but the former is also important, especially in Go, where the compiler is very sensitive to mutations (e.g., an unused import is a compiler error). For compilable mutants, we further distinguish between heuristics for equivalent mutants, killable mutants, and redundant mutants, as reported in Table 2.

Each of the four heuristic categories contains one or more distinct groups of rules, which in turn contain one or more related rules. For example, all rules that suppress mutants in logging statements (multiple rules for multiple types of logging statements and functions) form a distinct group because they all apply to logging, and the entire group aims to prevent unproductive killable mutants. The frequency indicates how often a category is applicable to a given changelist. For a detailed list of rules, please refer to the supplementary materials, which can be found online at http://doi.ieeecomputersociety.org/10.1109/TSE.2021.3107634.

3.2.1 Heuristics to Prevent Uncompilable Mutants

A mutant should be a syntactically valid program—otherwise, it would be detected by the compiler and would not add any value for testing. There are certain mutations, especially the ones that delete code, that violate this validity principle. A prime example is deleting code in Go; any unused variable or imported module produces a compiler error. The proposed heuristic gathers all used symbols and puts them in a container instead of deleting them so they remain referenced and the compiler is appeased.

TABLE 2
Arid Node Heuristics. Each Category Contains One or More Distinct Groups of One or More Related Rules

Category	Frequency	Distinct groups
Uncompilable	Common	1
Equivalent	Common	13
Unproductive killable	Very common	16
Redundant	Uncommon	2

3.2.2 Heuristics to Prevent Equivalent Mutants

Equivalent mutants, which are semantically equivalent to the mutated program, are a plague in mutation testing and cannot generally be detected automatically. However, there are some groups of equivalent mutants that can be accurately detected. For example, in Java, the specification for the `size` method of a `java.util.Collection` is that it returns a non-negative value. This means that mutations such as `collection.size() == 0` $\mapsto$ `collection.size() <= 0` are guaranteed to produce an equivalent mutant.

Another example for this category is related to memoization. Memoization is often used to speed up execution, but its removal inevitably causes the generation of equivalent mutants. The following heuristic is used to detect memoization: an `if` statement is a cache lookup if it is of the form `if a, ok := x[v]; ok return a`, i.e., if a lookup in the map finds an element, the `if` block returns that element (among other values, e.g., `Error` in Go). Such an `if` statement is a cache lookup statement and is considered arid by the *expert* function, as is its full body. The following example shows a cache lookup in Go:

```
var cache map[string]string
func get(key string) string {
  if val, ok := cache[key]; ok {
    return val
  }
  value := expensiveCalculation(key)
  cache[key] = value
  return value
}
```

Removing the `if` statement just removes caching, but does not change functional behavior, and hence yields an equivalent mutant. The program still produces the same output for the same input—albeit slower. Functional tests are not expected to detect such changes.

As a third example, a heuristic in this category avoids mutations of time specifications because unit tests rarely test for time, and if they do, they tend to use fake clocks. Statements invoking sleep-like functionality, setting deadlines, or waiting for services to become ready (like gRPC [25] server's `Wait` function that is always invoked in RPC servers, which are abundant in Google's code base) are considered arid by the *expert* function.

```
  sleep(100); rpc.set_deadline(10);
```

```
  sleep(200); rpc.set_deadline(20);
```

3.2.3 Heuristics to Prevent Unproductive Killable Mutants

Not all code is equally important: some code may result in killable mutants but the tests that kill them are not valuable and would not be written by experienced developers; such mutants are bad test goals. Examples of this category are increments of values in monitoring system frameworks, low level APIs or flag changes: these are easy to mutate, easy to test for, and yet mostly undesirable test goals.

A common way to implement heuristics in this category is to match function names; indeed we suppress mutants in calls to hundreds of functions, which is responsible for the largest proportion of suppressions by the *expert* function. The prime example of this category is a heuristic that marks any function call arid if the function name starts with the prefix `log` or the object on which the function is invoked is called `logger`. We validated this heuristic by randomly sampling 100 nodes that were marked arid by the `log` heuristic, and found that 99 indeed were correctly marked, while one had marginal utility. In total, we have accumulated fuzzy name suppression rules for more than 200 function families.

```
log.infof("network speed: %v", bytes/time)
```

```
log.infof("network speed: %v", bytes+time)
```

3.2.4 Heuristics to Prevent Redundant Mutants

Recall that the Mutation Testing Service generates at most one mutant per line and reports a restricted subset of surviving mutants during code review. Heuristics in this category suppress some mutants that are redundant (i.e., functionally equivalent to other mutants) for two reasons. First, while redundant mutants are functionally equivalent to one another, some of them are easier to reason about than others, rendering them as more productive. Second, when a developer updates their changelist, possibly writing tests to kill mutants, that change creates a new snapshot and triggers a rerun of the mutation service, thereby testing the change and possibly reporting new mutants. In order to improve developer productivity and user experience, the Mutation Testing Service should consistently generate the same mutant out of a pool of equally productive ones and avoid divergence from previously reported mutants, in particular for unchanged lines between snapshots. Such divergence would cause confusion, introduce cognitive overhead, and hence lower developer productivity.

As an example, in C++, the LCR mutation operator has a special case when dealing with `NULL` (i.e., `nullptr`), because of its logical equivalence with `false`:

ORIGINAL NODE		POTENTIAL MUTANTS

`if (x != nullptr)`	$\longmapsto$	`if (x)`
		`if (nullptr)`
		`if (x == nullptr)`
		`if (false)`
		`if (true)`

The mutants marked in bold are redundant because the value of `nullptr` is equivalent to `false`. Likewise, the opposite example, where the condition is `if (nullptr == x)`, yields redundant mutants for the left-hand side.

3.2.5 Experience With Heuristics

In our experience of applying heuristics, the highest productivity gains resulted from three heuristics implemented in the early days: suppression of mutations in logging statements, time-related operations (e.g., setting deadlines, timeouts, exponential backoff specifications etc.), and finally configuration flags. Most of the early feedback was about unproductive mutants in such code, which is ubiquitous in the code base. While it is hard to measure exactly, there is strong indication that these suppressions account for improvements in productivity from about 15% to 80%. Additional heuristics and refinements progressivley improved producitvity to 89%.

Heuristics are implemented by matching AST nodes with the full compiler information available to the mutation operator. Some heuristics are unsound: they employ fuzzy name matching and recognize AST shapes, but may suppress productive mutants. On the other hand, some heuristics make use of the full type information (like matching `java.util.HashMap::size` calls) and are sound. Sound heuristics are demonstrably correct, but we have had much more important improvements of perceived mutant usefulness from unsound heuristics.

4 MUTATION OPERATOR SELECTION STRATEGIES

After labeling arid nodes in the AST, the Mutation Testing Service generates mutants for the remaining, non-arid nodes. This involves two challenges. First, only generated mutants that survive the tests are reported to developers during code review; mutants that don't survive just use computational resources. Given that many mutants don't survive the tests and mutagenesis only generates a single mutant per line, the goal is to create mutants that have a high chance of survival. An iterative approach, where after the first round of tests further rounds of mutagenesis could be run for lines in which mutants were killed, would use the build and test systems inefficiently, and would take much longer because of multiple rounds. Similarly, generating all mutants per line is computationally too expensive. Second, not all surviving mutants are equally productive: depending on the context, certain mutation operators may produce better mutants than others. Therefore, the goal is to create surviving mutants that have a high chance of being productive. An effective mutation operator selection strategy not only constitutes a good trade-off between productivity and costs, but is also crucial for making mutation analysis results actionable during code review.

This section presents a basic random selection strategy that generates one mutant per covered line, considering information about arid nodes, and a targeted selection strategy, which additionally considers the past performance of mutation operators in similar context (Fig. 4).

4.1 Random Selection

A basic random line-based mutant selection approach could, for each line in a changelist, select one of the mutants that can be generated for that line uniformly at random. Alternatively, such an approach could randomly select a mutation point in that line first and then randomly select an applicable mutation operator.

Recall that our approach to mutation testing is based on the identification of arid nodes, which should not be mutated at all. Furthermore, our approach generates at most a single mutant per line; no additional mutants are ever generated. Listing 1 describes our random selection algorithm that accounts for these two design decisions. The mutation operators available for a given language are randomly shuffled and tried one by one, for each covered, changed line corresponding to non-arid nodes in the changelist, until a mutant is generated for that line or all operators have been tried. If multiple mutants can be generated in a line, only one mutant is generated, but which one depends on the random shuffle and the AST itself. For example, the ROR mutation operator cannot generate a mutant in a line that has no relational operators, but the SBR operator might—most lines can be deleted.

Listing 1. Random Selection With Suppression

```
function Mutagenesis(diff_ast)
  mutants ← Ø
  productive_ast = remove_arid_nodes(diff_ast)
  ops = shuffle({UOI, ROR, SBR, LCR, AOR})
  for line in covered_lines(productive_ast)
    for op in ops
      if can_generate(op, line)
          mutants ∪= generate_mutant(op, line)
          break
  return mutants
```

4.2 Targeted Selection

In contrast to the random selection, the targeted selection strategy ranks the mutation operators by their historical productivity considering the AST context, as shown in Listing 2.

Listing 2. Targeted Selection With Suppression

```
function Mutagenesis(diff_ast)
  mutants ← Ø
  productive_ast = remove_arid_nodes(diff_ast)
  ops = {UOI, ROR, SBR, LCR, AOR}
  for line in covered_lines(productive_ast)
    ranked_ops = historic_productivity_rank(line, ops)
    for op in ranked_ops
      if can_generate(op, line)
        mutants ∪= generate_mutant(op, line)
        break
  return mutants
```

The mutation operator ranking for a given AST node is based on historical information, in particular survivability and productivity. A mutation operator's *survivability* is the ratio of surviving mutants generated by that operator in a given context. A mutation operator's *productivity* is the ratio of productive mutants generated by that operator in a given context. Productivity is based on developer feedback: during code review authors and reviewers can flag mutants shown in a changelist as productive or unproductive. As

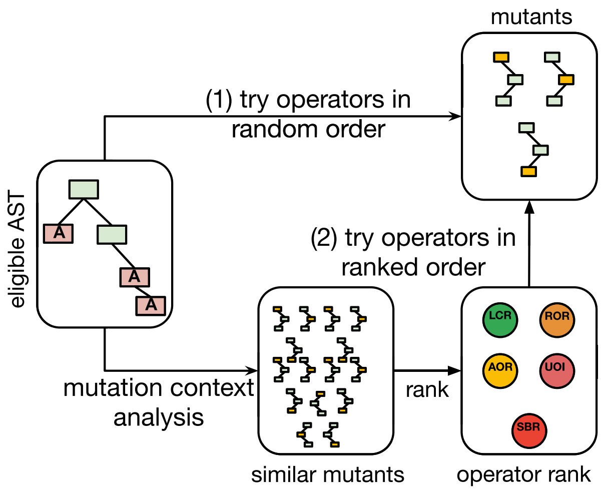

mutants
(1) try operators in random order
eligible AST
A
A
A
(2) try operators in ranked order
mutation context analysis
rank
similar mutants
operator rank
LCR
ROR
AOR
UOI
SBR

Fig. 4. Random (1) versus Targeted (2) mutation selection.

these developers understand the context of the mutants they are flagging, unlike participants performing a labeling task in a study, we consider this information a strong signal.

For each mutant, the AST context, which describes the environment of the AST node that was mutated, is stored along with the productivity feedback and whether the mutant was killed or not. The targeted selection strategy uses this information to identify AST nodes that are similar to the mutated one, based on the AST context. The historical information of the mutants generated for these similar AST nodes is then used to rank the mutation operators, rather than using a random order. Mutagenesis is then attempted in the resulting order to maximize the probability that the mutant will survive and will be productive.

4.3 Mutation Context

In order to apply historical information about survivability and productivity, we need to decide how similar candidate mutations are compared to past mutations. We define a mutation to be similar if it happened in a similar context, e.g., replacing a relational operator within an `if` condition that is the first statement in the body of a `for` loop, as shown in Listing 3.

Listing 3. C++ Snippet: An `if` Statement Within a `for` Loop

```
for (int i = 0; i < kMax; ++i) {
  if (i < kMax / 2) {
    return i / 2;
  } else {
    return i * 2;
  }
}
```

To efficiently capture the similarity of the context of two mutations, we use the hashing framework for tree-structured data introduced by Tatikonda *et al.* [26], which maps an unordered tree into a multiset of simple structures referred to as *pivots*. Each pivot captures information about the relationship among the nodes of the tree (see Section 4.4).

Finding similar mutation contexts is then reduced to finding similar pivot multisets. To identify similar pivot

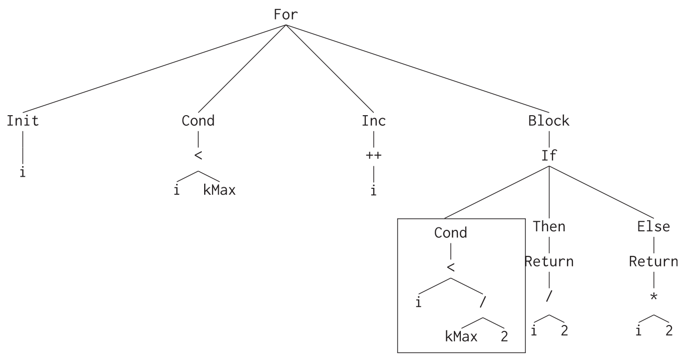


Fig. 5. AST for the C++ example in Listing 3.

multisets, we produce a MinHash [27] inspired fingerprint of the pivot multiset. Because the distance in the fingerprint space correlates with the distance in the tree space, we can find similar mutation contexts efficiently by finding similar fingerprints of nodes under mutation.

4.4 Generating Pivots From ASTs

In order to capture the intricate relationship between nodes in the AST, we translate the AST into a multiset of pivots. A pivot is a triplet of nodes from the AST that encodes their relationship; for nodes u and v, a pivot p is tuple (lca, u, v), where lca is the lowest common ancestor of nodes u and v. The pivot represents a subtree of the AST. The set of all pivots involving a particular node describes the tree from the point of view of that node. In mutation testing, we are only interested in nodes that are close to the node being mutated, so we constrain the set of pivots to pivots containing nodes that are a certain distance from the node considered for mutation.

In the example of replacing a relational operator in an `if` condition within a body of the `for` loop in Listing 3, one pivot might be (`if`, `Cond`, $*$), and another (`Cond`, i, `kMax`). All combinations of two nodes within some distance from the node being mutated in the AST in Fig. 5 and their lowest common ancestor make pivot structures.

Pivot multisets P precisely preserve the structural relationship of the tree nodes (*parent-child* and *ancestor* relations), so the tree similarity of two AST subtrees $T1$ and $T2$ can be measured as the Jaccard index of the pivot multisets [26] as shown in Equation (2).

$$d(T1, T2) = \text{Jaccard}(P(T1), P(T2)) = \frac{|P(T1) \cap P(T2)|}{|P(T1) \cup P(T2)|}. \tag{2}$$

4.5 Fingerprinting Pivot Multisets

Pivot multisets are potentially quadratic in tree size, leading to costly union and intersection operations. Even a trivial `if` statement with a single `return` statement produces large pivot sets, and set operations become prohibitive. To alleviate that, a fingerprinting function is applied to convert large pivot multisets into fixed-sized fingerprints.

We hash the pivot sets to single objects that form the multiset of representatives for the input AST. The size of the multiset can be large, especially for large programs. In order to improve the efficiency of further manipulation, we use a signature function that converts large pivot hash sets into shorter signatures. The signatures are later used to compute the similarity between the trees, taking into consideration only the AST node type and ignoring everything else, like type data or names of the identifiers.

We use a simple hash function to hash a single pivot $p = (lca, u, v)$ into a fixed-size value, proposed by Tatikonda and Parthasarathy [26].

$$h(p) = (a \cdot lca + b \cdot u + c \cdot v) \bmod K$$

$$a, b, c \in \mathbb{Z}_{\mathbb{P}}.$$

For a, b, c we pick small primes, and for K a large prime that fits in 32 bits. To be able to hash AST nodes, we assign sparse integer hash values to different AST node types in each language, e.g., a C++ `FunctionDecl` is assigned 8500, and `CXXMethodDecl` 8,600. For nodes in the pivot (lca, u, v) we use these assigned hashes.

For example, given $a = 17$, $b = 59$, $c = 83$ and $K = 15485863$, we can calculate the hash of the pivot (`if`, $<$, $*$), as simply as

$$\begin{aligned} h1 &= (59 * 32800 + 17 * 22400 + 83 * 22400)\% \ 15485863 \\ &= 4175200, \end{aligned}$$

with 32,800 and 22,400 being the integer hash values assigned to `IfStmt` and `BinaryOperator` C++ AST nodes.

The signature for such a bag of representatives is generated using a MinHashing technique. The set of pivots is permuted and hashed under that permutation. To minimize the false positives and negatives (i.e., different trees hash to similar hashes, or vice versa), this is repeated k times, resulting in k-MinHashes.

The goal is that the signatures are similar for similar (multi)sets and dissimilar for dissimilar ones. Jaccard similarity between two sets can be estimated by comparing their MinHash signatures in the same way [27], as shown in Equation (3). The MinHash scheme can be considered an instance of locality-sensitive hashing, in which ASTs that have a small distance to each other are transformed into hashes that preserve that property.

$$d(T1, T2) = \frac{|P(T1) \cap P(T2)|}{|P(T1) \cup P(T2)|} \approx \frac{|hash(T1) \cap hash(T2)|}{|hash(T1) \cup hash(T2)|}. \tag{3}$$

When mutating a node, we calculate its pivot set and hash it. We find similar AST contexts using nearest neighbor search algorithms. We observe how different mutants behave in this context and which mutation operators produce the most productive and surviving mutants. This is the basis for targeted mutation selection.

5 Evaluation

In order to bring value to developers, the Mutation Testing Service at Google needs to report few productive mutants, selected from a large pool of mutants—most of which are unproductive. Recall that a productive mutant elicits an effective test, or otherwise advances code quality [11]. Therefore, our goal is two-fold. First, we aim to select

TABLE 3
Summary of the Mutant Dataset. (Note That SQL, Common Lisp, and Kotlin are Excluded From our Analyses Because of Insufficient Data)

Language	Generated mutants			Survivability
	Count	Ratio	Per CL	
C++	7,197,069	42.5%	23.2	12.5%
Java	2,894,772	17.1%	14.8	13.2%
Go	1,988,798	11.7%	27.6	12.5%
Python	1,689,382	10.0%	21.3	13.2%
TypeScript	1,006,531	5.9%	20.8	10.8%
JavaScript	908,014	5.4%	31.0	9.4%
Dart	581,109	3.4%	17.4	16.3%
SQL	478,975	2.8%	91.2	11.7%
Common Lisp	148,289	0.9%	179.3	2.2%
Kotlin	42,209	0.2%	20.7	11.0%
Total	16,935,148	100%	21.8	12.5%

mutants with a high survival rate and productivity to maximize their utility as test objectives. Second, we aim to report very few mutants to reduce computational effort and avoid overwhelming developers with too many findings.

Since applying mutation testing on the entire code base is simply infeasible, we focus on diff-based mutation in our evaluation. In addition to the basic design decision of applying mutation testing at the level of changelists, two technical solutions reduce the number of mutants: (1) mutant suppression using arid nodes and (2) one-per-line mutant selection. Our evaluation uses two datasets (Section 5.1) and answers four research questions. The first research question concerns the effectiveness of our two technical solutions:

- *RQ1 Mutant suppression*. How effective is mutant suppression using arid nodes and 1-per-line mutant selection? (Section 5.2)

To understand the influence of mutation operator selection on mutant survivability and productivity in the remaining non-arid nodes, we consider historical data, including developer feedback. We aim to answer the following two research questions:

- *RQ2 Mutant survivability*. Does mutation operator selection influence the probability that a generated mutant survives the test suite? (Section 5.3)
- *RQ3 Mutant productivity*. Does mutation operator selection influence developer feedback on a generated mutant? (Section 5.4)

Having established the influence of individual mutation operators on survivability and productivity, the final question is whether mutation context can be used to improve both. Therefore, our final research question is as follows:

- *RQ4 Mutation context*. Does context-based selection of mutation operators improve mutant survivability and productivity? (Section 5.5)

5.1 Experiment Setup

For our analyses, we established two datasets, one with data on all mutants, and one containing additional data on mutation context for a subset of all mutants.

TABLE 4
Number of Mutants per Mutation Operator

Operator	Generated mutants		Survivability
	Count	Ratio	
SBR	11,522,932	68.0%	12.7%
UOI	3,137,375	18.5%	9.6%
LCR	1,305,499	7.7%	16.3%
ROR	672,009	4.0%	14.7%
AOR	297,333	1.8%	13.5%
Total	16,935,148	100%	12.5%

Mutant Dataset. The mutant dataset contains 16,935,148 mutants across 10 programming languages: C++, Java, Go, Python, TypeScript, JavaScript, Dart, SQL, Common Lisp, and Kotlin. Table 3 summarizes the mutant dataset and gives the number and ratio of mutants per programming language, the average number of mutants per changelist and the percentage of mutants that survive the test suite. Table 4 breaks down the numbers by mutation operator.

We created this dataset by gathering data on all mutants that the Mutation Testing Service generated since its inauguration, which refers to the date when we made the service broadly available, after the initial development of the service and its suppression rules (see Section 3.2.5). We did not perform any data filtering, hence the dataset provides information about all mutation analyses that were run.

In total, our data collection considered 776,740 changelists that were part of the code review process. For these, 16,935,148 mutants were generated, out of which 2,110,489 were reported. Out of all reported mutants, 66,798 received explicit developer feedback. For each considered changelist, the mutant dataset contains information about:

- affected files and affected lines,
- test targets testing those affected lines,
- mutants generated for each of the affected lines,
- test results for the file at the mutated line, and
- mutation operator and context for each mutant.

Our analysis aims to study the efficacy and perceived productivity of mutants and mutation operators across programming languages. Note that our mutant dataset is likely specific to Google's code style and review practices. However, the code style is widely adopted [28], and the modern code review process is used throughout the industry [29].

Information about mutant survivability per programming language or mutation operator can be directly extracted from the dataset and allows us to answer research questions *RQ1*, *RQ2* and *RQ3*.

Context Dataset. The context dataset contains 4,068,241 mutants (a subset of the mutant dataset) for the top-four programming languages: C++, Java, Go, and Python. Each mutant in this dataset is enriched with the information of whether our context-based selection strategy would have selected that mutant. When generating mutants, we would also run the context-based prediction, and we persisted the prediction information along with the mutants. If the randomly chosen operator was indeed what the prediction service picked, this mutant is the one with the highest predicted value. For each mutant, the dataset contains:

- all information from the mutant dataset,
- predicted survivability and productivity for each mutation in similar context, and
- information about whether the mutant has the highest predicted survivability/productivity.

We created this dataset by using our context-based mutation selection strategy during mutagenesis on all mutants during a limited period of time. During this time, we automatically annotated the mutants, indicating whether a mutant would be picked by the context-based mutation selection strategy along with the mutant outcome in terms of survivability and productivity. This dataset enables the evaluation of our context-based mutation selection strategy and allows us to answer research question *RQ4*.

Experiment Measures. Surviving the initial test suite is a precondition for surfacing a mutant, but survivability alone is not a good measure of mutant productivity. Developer feedback indicating that a mutant is indeed (un)productive is a stronger signal.

We measure mutant productivity via user feedback gathered from Critique (Section 2.4), where each reported mutant displays a *Please fix* (productive mutant) and a *Not useful* (unproductive mutant) link. *Please fix* corresponds to a request to the author of a changelist to improve the test suite based on the reported mutant; *not useful* corresponds to a false alarm or generally a non-actionable code finding. 82% of all reported mutants with feedback were labeled as productive by developers. Note that this ratio is an aggregate over the entire data set. Since the inauguration of the Mutation Testing Service, productivity has increased over time from 80% to 89% because we generalized the feedback on unproductive mutants and created suppression rules for the *expert* function, described in Section 3. This means that later mutations of nodes in which mutants were found to be unproductive will be suppressed, generating fewer unproductive mutants over time. Reported mutants without explicit developer feedback are not considered for the productivity analysis.

5.2 RQ1 Mutant Suppression

In order to compare our mutant-suppression approach with the traditional mutagenesis, we (1) randomly sampled 5,000 changelists from the mutant dataset, (2) determined how many mutants traditional mutagenesis produces, and (3) compared the result with the number of mutants generated by our approach. (Since traditional mutation analysis is prohibitively expensive at scale, we adapted our system to only generate all mutants for the selected changelists.) Fig. 6 shows the results for three strategies: no suppression (traditional), select one mutant per line, and select one mutant per line after excluding arid nodes (our approach). We include the 1-per-line approach in the analysis to evaluate the individual contribution of the arid-node suppression, beyond sampling one mutant per line.

As shown in Table 5, the median number of generated mutants is 820 for traditional mutagenesis, 77 for 1-per-line selection, and only 7 for arid-1-per-line selection. Hence, our mutant-suppression approach reduces the number of mutants by two orders of magnitude. Table 5 also shows

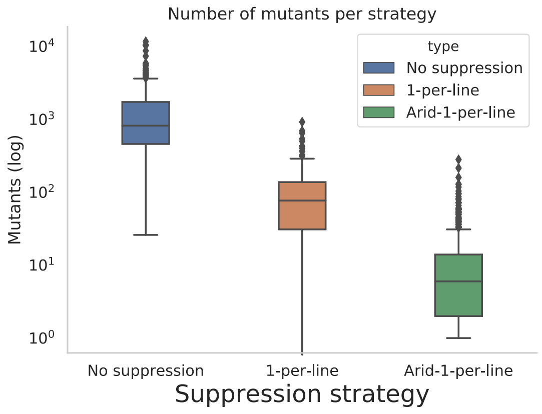


Fig. 6. Number of generated mutants per changelist for no suppression (traditional mutagenesis), 1-per-line and arid-1-per-line (our approach). (Note the log-scaled vertical axis.)

the results for a Mann-Whitney U test, which confirms that the distributions are statistically significantly different.

Our mutant-suppression approach generates fewer than 20 mutants for most changelists; the 25th and 75th percentiles are 3 and 19, respectively. In contrast, the 25th and 75th percentiles for 1-per-line are 31 and 138 mutants. Traditional mutagenesis generates more than 450 mutants for most changelists (the 25th and 75th percentiles are 460 and 1734, respectively), further underscoring that this approach is impractical, even at the changelist level. Presenting hundreds of mutants, most of which are not actionable, to a developer would almost certainly result in that developer abandoning mutation testing altogether.

RQ1. *Arid-node suppression and 1-per-line selection significantly reduce the number of mutants per changelist, with a median of only 7 mutants per changelist (compared to 820 mutants for traditional mutagenesis).*

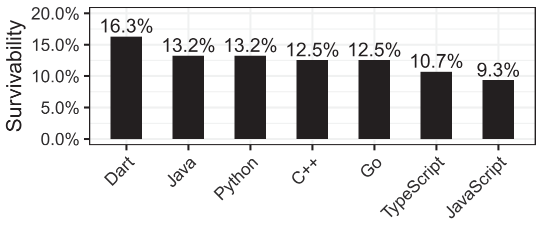
5.3 RQ2 Mutant Survivability

Mutant survivability is important because we generate at most a single mutant per line—if that mutant is killed, no other mutant is generated. To be actionable, mutants have to be reported as soon as possible in the code review process, as described in Section 4. Therefore, we aim to maximize mutant survivability because it directly impacts the number of reported mutants.

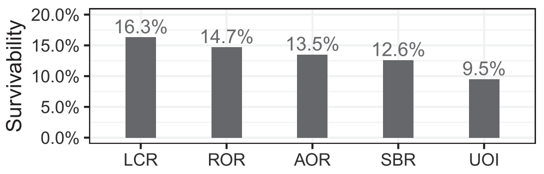
Overall, 87.5% of all generated mutants are killed by the initial test suite. Note that this is not the same as the traditional mutation score [30] (ratio of killed mutants to the total number of mutants) because mutagenesis is probabilistic and only generates a subset of all mutants. This means only

TABLE 5
Mann-Whitney U Test Comparing the Distributions of the Number of Mutants Generated by Different Strategies

STRATEGY A	STRATEGY B	P-VALUE	MEDIAN A	MEDIAN B
No suppression	1-per-line	<.0001	820	77
1-per-line	Arid-1-per-line	<.0001	77	7
No suppression	Arid-1-per-line	<.0001	820	7

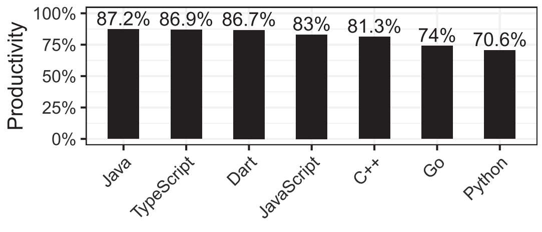


(a) Survivability per programming language.

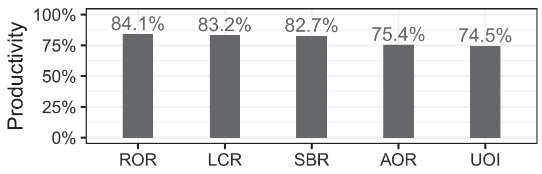


(b) Survivability per mutation operator.

Fig. 7. Mutant survivability.



(a) Productivity per programming language.



(b) Productivity per mutation operator.

Fig. 8. Mutant productivity.

a fraction of all possible mutants are generated and evaluated, and many other mutants are never generated because they are associated with arid nodes.

Tables 3 and 4 show the distribution of number of mutants and mutant survivability, broken down by programming language and mutation operator. Fig. 7 visualizes the mutant survivability data. Because the SBR mutation operator can be applied to almost any non-arid node in the code, it is no surprise that this mutation operator dominates the number of mutants, contributing roughly 68% of all mutants. While SBR is a prolific and versatile mutation operator, it is also the second least likely to survive the test suite: when applicable to a changelist, SBR mutants are reported during code review with a probability of 12.6%. Overall, mutant survivability is similar across mutation operators, with a notable exception of UOI, which has a survivability of only 9.5%. Mutant survivability is also similar across programming languages with the exception of Dart, whose mutant survivability is noticeably higher. We conjecture that this is because Dart is mostly used for web development which has its own testing challenges.

RQ2. *Different mutation operators result in different mutant survivability; for example, the survival rate of LCR is almost twice as high as that of UOI.*

5.4 RQ3 Mutant Productivity

Mutant productivity is the most important measure, because it directly measures the utility of a reported mutant. Since we only generate a single mutant in a line, that mutant ideally should not just survive the test suite but also be productive, allowing developers to improve the test suite or the source code itself. Given Google's high accuracy and actionability requirements for surfacing code findings during code reviews, we rely on developer feedback as the best available measure for mutant productivity. Specifically, we consider a mutant a developer marked with *Please fix* to be more productive than others. Likewise, we consider a mutant a developer marked with *Not useful* to be less productive than others. We compare the mutant productivity across mutation operators and programming languages.

Fig. 8 shows the results, indicating that mutant productivity is similar across mutation operators, with AOR and UOI mutants being noticeably less productive. For example, ROR mutants are productive 84.1% of the time, whereas, UOI mutants are only productive 74.5% of the time. The differences between programming languages are even more pronounced, with Java mutants being productive 87.2% of the time, compared to Python mutants that are productive 70.6% of the time. This could be due to code conventions, language common usecase scenarios, testing frameworks or simply the lack of heuristics. We have found that Python code generally requires more tests because of the lack of the compiler. Unlike Python which is mostly used for backends, JavaScript, TypeScript and Dart are predominantly used in frontend code that is radically different.

RQ3. *ROR, LCR, and SBR mutants show similar productivity, whereas AOR and UOI mutants show noticeably lower productivity.*

5.5 RQ4 Mutation Context

We examine whether context-based selection of mutation operators improves mutant survivability and productivity. Specifically, we determine whether context-based selection of mutation operators increases the probability of a generated mutant to survive and to result in a *Please fix* request, when compared to the random-selection baseline.

Fig. 9 shows that selecting mutation operators based on the AST context of the node under mutation substantially increases the probability of the generated mutant to survive and to result in a *Please fix* request. While improvements vary across programming languages and across mutation operators, the context-based selection consistently outperforms random selection. The largest productivity improvements are achieved for UOI, AOR, and SBR, which generate most of all mutants. Intuitively, these improvements mean that context-based selection results in twice as many productive UOI mutants (out of all generated mutants), when compared to random selection. Fig. 9 also shows to what extent these improvements can be attributed to the fact that

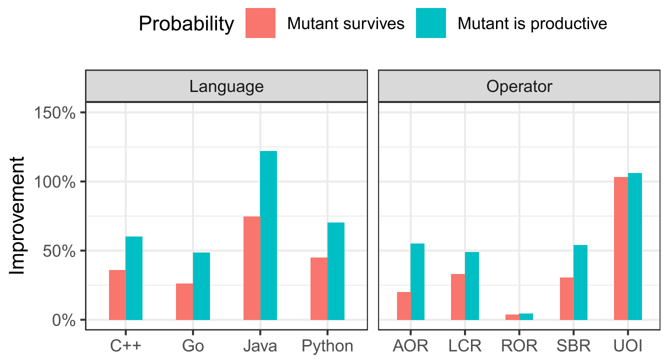


Fig. 9. Improvements achieved by context-based selection. (0% improvement corresponds to random selection.)

simply more mutants survive. Since the improvements for productivity increase even more than those for survivability, context-based selection not only results in more reported mutants but also in higher productivity of these mutants. Overall, the survival rate increases by over 40% and the probability that a reviewer asks for a generated mutant to be fixed increases by almost 50%.

It is important to put these improvements into context. Probabilistic diff-based mutation analysis aggressively trims down the number of considered mutants from thousands in a representative file to a mere few, and enables mutants to be effectively presented to developers as potential test targets. The random-selection approach produces fewer surviving mutants of lower productivity.

RQ4. *Context-based selection improves the probability that a generated mutant survives by more than 40% and the probability that a generated mutant is productive by almost 50%.*

6 Related Work

There are several veins of research that are related to this work. Just *et al.* proposed an AST-based program context model for predicting mutant effectiveness [31]. Fernandez *et al.* developed various rules for Java programs to detect equivalent and redundant mutants [32]. The initial results are promising for developing selection strategies that outperform random selection. Further, Zhang *et al.* used machine learning to predict mutation scores, both on successive versions of a given project, and across projects [33]. Finally, the PIT project makes mutation testing usable by practicing developers and has gained adoption in the industry [16].

There has been a lot of focus on computational costs and the equivalent mutant problem [34]. There is much focus on avoiding redundant mutants, which leads to increase of computational costs and inflation of the mutation score [35], and instead favoring hard-to-detect mutants [36], [37] or dominator mutants [38]. Mutant subsumption graphs have similar goals but mutant productivity is much more fuzzy than dominance or subsumption.

Effectiveness for mutants is primarily defined in terms of redundacy and equivalence. This approach fails to consider the notion that non-reduntant mutants might be unproductive or that equivalent mutants can be productive [39]. From our experience, reporting equivalent mutants has been a vastly easier problem than reporting unproductive non-reduntant and non-equivalent mutants.

Our approach for targeted mutant selection (Section 4) compares the context of mutants using tree hashes. The specific implementation was driven by the need for consistency and efficiency, in order to make it possible to look up similar AST contexts in real time during mutant creation. In particular, the hash distances need to be preserved over time to improve the targeted selection. There are approaches to software clone detection [40] that similarly use tree-distances (e.g., [41], [42], [43], [44], [45]). Whether alternative distance measurements can be scaled for application at Google and whether they can further improve the targeted selection remains to be determined in future work.

This approach is similar to tree-based approaches in software clone detection [40], which aims to detect that a code fragment is a copy of some original code, with or without modification. The AST-based techniques can detect additional categories of modifications like identifier name changes or type aliases, that token-based detection cannot, and the insensitivity of to variable names is important for the mutation context. However, clone detection differs drastically in its goal: it cares about detecting code with the same semantics, in spite of the syntactical changes made to it. While clone detection might want to detect that an algorithm has been copied and then changed slightly, e.g., a recursion rewritten to an equivalent iterative algorithm, mutation testing context cares only about the neighboring AST nodes: in the iterative algorithm, the most productive mutants will be those that thrived before in such code, not the ones that thrived for a recursive algorithm. In order to look up similar AST contexts in real time, as mutants are created, we require a fast method that preserves hash distance over time. For these consistency and efficiency reasons, we opted for the described tree-hashing approach.

7 Conclusion

Mutation testing has the potential to effectively guide software testing and advance software quality. However, many mutants represent unproductive test goals; writing tests for them does not improve test suite efficacy and, even worse, negatively affects test maintainability.

Over the past six years, we have developed a scalable mutation testing approach and mutant suppression rules that increased the ratio of productive mutants, as judged by developers. In the early phases of the project, the initial mutant suppression rules improved the ratio of productive mutants from 15% to 80%. As the product matured, additional mutant suppression rules improved the productivity to 89%. Three strategies were key to success. First, we devised an incremental mutation testing strategy, reporting at most one mutant per line of code—targeting lines that are changed and covered. Second, we have created a set of rule-based heuristics for mutant suppression, based on developer feedback and manual analyses. Third, we devised a probabilistic, targeted mutant selection approach that considers mutation context and historical mutation results.

Given the success of our mutation testing approach and the positive developer feedback, we expect that further adoption by development teams will result in additional refinements of the suppression and selection strategies. Furthermore, an important aspect of our ongoing research is to understand the long-term effects of mutation testing on developer behavior [20].

References

[1] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at Google," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 955–963.

[2] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," George Mason Univ., Fairfax, VA, Tech. Rep. ISSE-TR-96–100, 1996.

[3] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Proc. 4th IEEE Int. Conf. Softw. Testing Verification Validation*, 2011, pp. 90–99.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[6] 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 *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 654–665.

[7] Y. T. Chen *et al.*, "Revisiting the relationship between fault detection, test adequacy criteria, and test set size," in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2020, pp. 237–249.

[8] J. Micco, "The state of continuous integration testing at Google," 2017. [Online]. Available: https://ai.google/research/pubs/pub45880

[9] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *Commun. ACM*, vol. 59, pp. 78–87, 2016.

[10] D. Schuler and A. Zeller, "(Un-)covering equivalent mutants," in *Proc. 3rd Int. Conf. Softw. Test. Verification Validation*, 2010, pp. 45–54.

[11] G. Petrović, M. Ivanković, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation Workshops*, 2018, pp. 47–53.

[12] G. Petrovic and M. Ivankovic, "State of mutation testing at Google," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 163–171.

[13] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.

[14] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation Testing for the New Century*, Berlin, Germany: Springer, 2001, pp. 34–44.

[15] Bazel build system, 2015. [Online]. Available: https://bazel.io/

[16] H. Coles, "Real world mutation testing," Accessed: Jul. 2021. [Online]. Available: http://pitest.org

[17] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *Proc. IEEE 23rd Int. Symp. Softw. Rel. Eng.*, 2012, pp. 11–20.

[18] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 433–436.

[19] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 612–615.

[20] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does mutation testing improve testing practices?," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 910–921.

[21] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[22] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Conf.*, 2015, pp. 598–608.

[23] M. Ivankovic, G. Petrovic, R. Just, and G. Fraser, "Code coverage at Google," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 955–963.

[24] S. S. Muchnick, *Advanced Compiler Design Implementation*. Burlington, MA, USA: Morgan Kaufmann, 1997.

[25] G. Inc., "gRPC: A high performance, open-source universal RPC framework," 2006. [Online]. Available: https://grpc.io

[26] S. Tatikonda and S. Parthasarathy, "Hashing tree-structured data: Methods and applications," in *Proc. 2010 IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 429–440.

[27] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *J. Comput. Syst. Sci.*, vol. 60, no. 3, pp. 630–659, 2000.

[28] Google style guides. Accessed: Jul. 2021. [Online]. Available: https://google.github.io/styleguide/

[29] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 712–721.

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[31] R. Just, R. J. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proc. Int. Symp. Softw. Testing Anal.*, 2017, pp. 284–294.

[32] L. Fernandes *et al.*, "Avoiding useless mutants," in *Proc. 16th ACM SIGPLAN Int. Conf. Generative Program.: Concepts Experiences*, 2017, pp. 187–198.

[33] J. Zhang *et al.*, "Predictive mutation testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2016, pp. 342–353.

[34] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.

[35] R. Just and F. Schweiggert, "Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators," *Softw. Testing Verification Rel.*, vol. 25, no. 5/7, pp. 490–507, 2015.

[36] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 919–930.

[37] W. Visser, "What makes killing a mutant hard," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 39–44.

[38] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proc. IEEE 7th Int. Conf. Softw. Testing Verification Validation*, 2014, pp. 21–31.

[39] P. McMinn, C. J. Wright, C. J. McCurdy, and G. Kapfhammer, "Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 427–463, May 2019.

[40] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School Comput. TR*, vol. 541, no. 115, pp. 64–68, 2007.

[41] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Softw. Maintenance*, 1998, pp. 368–377.

[42] W. Yang, "Identifying syntactic differences between two programs," *Softw., Pract. Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[43] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 96–105.

[44] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Proc. 4th IEEE Int. Workshop Source Code Anal. Manipulation*, 2004, pp. 128–135.

[45] W. S. Evans, C. W. Fraser, and F. Ma, "Clone detection via structural abstraction," *Softw. Quality J.*, vol. 17, no. 4, pp. 309–330, 2009.