# State of Mutation Testing at Google

Goran Petrović
Google Inc.
goranpetrovic@google.com

Marko Ivanković
Google Inc.
markoi@google.com

## ABSTRACT

Mutation testing assesses test suite efficacy by inserting small faults into programs and measuring the ability of the test suite to detect them. It is widely considered the strongest test criterion in terms of finding the most faults and it subsumes a number of other coverage criteria. Traditional mutation analysis is computationally prohibitive which hinders its adoption as an industry standard. In order to alleviate the computational issues, we present a diff-based probabilistic approach to mutation analysis that drastically reduces the number of mutants by omitting lines of code without statement coverage and lines that are determined to be uninteresting - we dub these *arid lines*. Furthermore, by reducing the number of mutants and carefully selecting only the most interesting ones we make it easier for humans to understand and evaluate the result of mutation analysis. We propose a heuristic for judging whether a node is arid or not, conditioned on the programming language. We focus on a code-review based approach and consider the effects of surfacing mutation results on developer attention. The described system is used by 6,000 engineers in Google on all code changes they author or review, affecting in total more than 13,000 code authors as part of the mandatory code review process. The system processes about 30% of all diffs across Google that have statement coverage calculated. About 15% of coverage statement calculations fail across Google.

**ACM Reference Format:**
Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3183519.3183521

## 1 INTRODUCTION

Software testing is a widely used technique for ensuring software quality. To measure test suites' effectiveness in detecting faults in software, various methods are available in the industry. Code coverage is used at Google as one such measure. However, coverage alone might be misleading, as in many cases where statements are covered but their consequences not asserted upon [15]. Mutation analysis inserts systematic faults (mutations) into the source code under test producing mutants of the original code, and judges the effectiveness of the test suite by its ability to detect those faults. Mutation analysis is widely considered the best method of evaluating test suite efficacy [1]. Mutants resemble real world bugs, and that the test suite effectiveness in detecting mutants is correlated to its effectiveness in detecting real faults [11]. While being a powerful tool, it is often computationally prohibitive. Furthermore, the cost of developer attention in evaluating and mitigating the results of the analysis is also expensive and can be exorbitant even for medium sized systems.

To leverage mutation analysis in a large complex software system like Google's, in this work we propose a diff-based approach to mutation analysis. Moreover, in the work we describe a method of transitive mutation suppression of uninteresting, arid lines based on developer feedback and program's AST. A diff-based approach greatly reduces the number of lines in which mutants are created, and the suppression of arid lines cuts the number of potential mutants further; combined, these two approaches make mutation analysis feasible even for colossal complex systems (Google's monolithic repository contains approximately 2 billion lines of code [16]). The contributions of this work are as follows:

- We propose a scalable mutation analysis framework integrated with the code review process. The approach hinges on mutation suppresion in arid nodes based on developer feedback.
- We empirically validated the proposed approach on the Google codebase by evaluating more than 70'000 diffs, testing 1.1 million mutants and surfacing 150'000 actionable findings during code review. Using this developer feeback loop, the reported usefulness of the surfaced results improved from 20% to 80%.

## 2 PROBLEM STATEMENT AND BACKGROUND

### 2.1 Mutation testing

Mutation testing, a process of inserting small faults into programs and measuring test suite effectiveness of detecting them was originally proposed by DeMillo et al. [3]. Various mutation operators that define transformations of the program were defined over time. Each transformation results in a new program, called *mutant*, that differs slightly from the original. The process of creating a mutant from the original program is called *mutagenesis*. Test suite efficacy is measured by its ability to detect those mutants. Mutants for which at least one test in the test suite fails are dubbed detected or killed. Consequently, mutants that are not detected by the test suite are called living mutants. The more mutants it kills, the more effective the test suite is in detecting faults.


*ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5659-6/18/05.
https://doi.org/10.1145/3183519.3183521

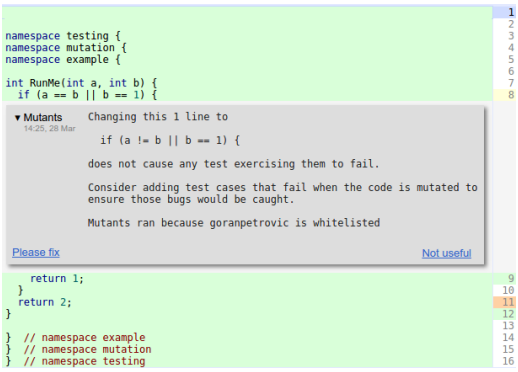



**Figure 1: Mutant finding shown in the Critique - Google code review tool**

Mutation score is the ratio of killed mutants to the total number of mutants and is a measure of this efficacy.

Selective mutation can substantially reduce the cost of mutation analysis [14], but alone is not enough to make mutation analysis scale for Google's development workflow.

## 2.2 Mutation score

The Google repository contains about two billion lines of code [16]. On average, 40'000 changes are commited to the codebase every workday, roughly 16'000 changes are authored by human authors and 24'000 by automated systems. At present it is infeasably expensive to compute the absolute mutation score for the codebase at any given fixed point. It would be even more expensive to keep re-computing the mutation score in any fixed time period (e.g., daily or weekly) and it is almost imposible to compute the full score after each commit. In addition to the computation costs of the mutation score, we were also unable to find a good way to surface it to the engineers in an actionable way.

## 2.3 Developer tools

*2.3.1 Code review process.* Most changes to Google's monolithic codebase, except for a limited number of fully automated changes, are reviewed by engineers before they are merged into the source tree. Potvin and Levenberg [16] provide a comprehensive overview of Google's development ecosystem. The review process is the cornerstone of the quality of the codebase, stated humorously by Linus' Law [17]: "Given enough eyeballs, all bugs are shallow". Reviewers can leave comments on the changed code that must be resolved by the author. A special type of comment generated by an automated diff analyzer is known as a *finding*. The mutation analysis result is one example of an automated finding.

If an automated diff analyzer finding (e.g. a living mutant) is not 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 diff author with a single click, as shown in Figure 1. This feedback is accessible to the owner of the tool that created the findings, so quality metrics can be tracked and unhelpful findings triaged, and ideally prevented in the future.

Many analyzers are run automatically when a diff is sent for review, from linters and formatters to static code and build dependency analyzers, mostly based on the Tricorder code analysis platform [18]. Additionally, reviewers will also post their comments to the diff. As a result, surfacing non-actionable findings during code review has a negative impact on the author and the reviewers.

We argue that the code review process is the best location for surfacing changed code metrics because it maximizes the probability that the change will be acted upon, thus improving the test suite efficacy.

*2.3.2 Explicit build dependencies and coverage.* Google uses Bazel as its build system [9]. Build targets list their sources and dependencies explicitly. Test targets can contain multiple tests, and each test suite can contain multiple test targets. Using the explicit dependency and source listing, test coverage analysis provides information about which test target covers which line in the source code. Tests are executed in parallel.

Statement coverage analysis results link lines of code to a set of tests covering them. Line level coverage is used for test execution phase, where the minimal set of tests is run in the attempt to kill the mutant.

During code reviews, *absolute* and *incremental* code coverage is surfaced to the developers. Absolute 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 artefacts like comments or pure whitespace lines are not applicable for testing. Incremental coverage is the ratio of the number of lines covered by tests in the added or modified lines in the diff to the total number of added or modified lines in the diff.

# 3 PROBABILISTIC DIFF-BASED MUTATION TESTING ANALYSIS

Integrating analysis results into the existing developer workflow is crucial to making it effective [10]. If developers are compelled to execute a separate binary and act on its output, the usability of the tool is drastically reduced. For this reason, the results of the mutation analysis, e.g. living mutants, are surfaced during a code review as code findings as described by Sadowski et al. [18].

Google has a massive codebase, counting two billion lines of code [16] in various programming languages. The coverage distribution per project is shown in Figure 2. Although the statement coverage of most projects is satisfactory, the number of living mutants per diff is significant (median is 2 mutants, $99^{\text{th}}$ percentile is 43 mutants). To be of any use to the author and the reviewers, code findings need to be surfaced quickly, before the review is complete. The cost of executing tests for all mutants is prohibitive, therefore, new techniques are needed.

**Probabilistic.** For each line, at most one mutant is generated. Surfacing multiple mutants for a single line clutters the

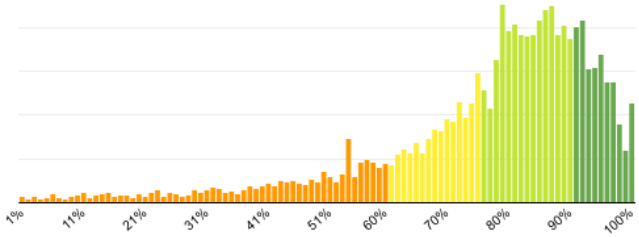



**Figure 2: Distribution of project statement coverage**

code review interface and looks confusing. If a mutant can be generated in a line, it will be. The mutation operator is picked at random from the set of applicable operators which depends on the context of the mutation. We elaborate on future work of leveraging the mutation context and historic feedback to predict the quality of the mutant in Section 6.

**Diff-based.** For each diff under review, mutation analysis is executed as soon as the incremental coverage analysis is complete. Incremental coverage analysis results are used to minimize the set of viable lines that should be mutated. Mutations are also suppressed in uninteresting, arid lines, described in Section 4. Only lines affected by the diff under review that are covered and are not arid are mutated. This way, only relevant lines that are interesting to both author and reviewers are analyzed.

The mutation testing service implements mutagenesis for seven programming languages: C++, Java, Python, Javascript, Go, TypeScript and Common Lisp. For each language, AOR, LCR, ROR, SBR and UOI mutators described in Mothra are implemented, shown in Figure 3 [13]. The ABS (absolute value insertion) mutator was reported predominantly not useful, mostly because it acted on time-and-count related expressions that are positive and nonsensical if negated, and is not used. We argue this is due to the style and features of our codebase, and is not applicable generally. For each file in the diff, a set of mutants is requested, one for each affected covered line. Affected lines are added or modified lines in the diff, and the covered lines are defined by the coverage analysis results, as described in Section 2.3.2. Mutagenesis is performed by traversing the Abstract Syntax Trees (AST) in each of the languages, and decisions are often done on the AST node level because it allows for fine-grained decisions due to the amount of context available.

# 4 ARID NODE DETECTION VIA ABSTRACT SYNTAX TREE TRAVERSAL

Some parts of code are less interesting than others. Surfacing living mutants in uninteresting statements like logging has a negative impact on human time spent analyzing the finding, and its cognitive overhead. Because adding test cases to kill mutants in uninteresting nodes is not viewed as improving the overall efficacy of the suite to detect faults, such mutants tend to survive. This section proposes an approach for mutation suppression and a heuristic for detecting such AST nodes in which mutation is to be suppressed. There is a tradeoff between the correctness and usability of results; the proposed heuristic may suppress mutation in relevant nodes as a side-effect of reducing uninteresting node mutations. We argue that this is a good tradeoff because the number of possible mutants is always orders of magnitude larger than what we could reasonably present to the developers within the existing developer tools, and it's more effective to prevent high impact faults, rather than arid faults.

## 4.1 Arid nodes

When parsing the source code to the AST, compilers construct a tree of nodes. Nodes may be statements, expressions or declarations and they are connected with the child-parent relationships into a tree to represent their connections in the source code [12]. Most compilers diferentiate simple and compound nodes. Simple nodes have no other nodes as part of their body, e.g. a call expression names a function and parameters, but has no body. Compound nodes have at least one body, e.g. a `for` loop might have a body, while an `if` statement might have two: `then` and `else` branches. An example of an arid node would be a log statement, calls to memory-reserving functions like `std::vector::reserve` or writes to `stdout`: scenarios tipically not tested by unit tests.

The heuristic for labeling nodes as arid is two-fold and is defined in equation 1. The first part operates on simple nodes and is represented by an expert curated manually for each programming language and is adjusted over time. The second part operates on compound nodes, and is defined recursively. A compound node is an arid node iff all of its compounds are arid. There are different categories of arid nodes:

- nodes are associated with logging, testing,
- nodes that control non-functional properties, and
- node that are axiomatic for the language and where mutant would be trivially killed by higher order testing.

**Expert**. Let $N \in T$ be a node in the abstract syntax tree $T$ of a program. Let *simple* be a boolean function determining whether a node is simple (compound nodes contain their children nodes). Let *expert* be a boolean function over a subset of simple statements in $T$ encoding manually curated knowledge on arid simple nodes. Then,

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

The *expert* function that flags simple nodes as arid is developed over time to incorporate developer feedback on reported 'Not useful' mutants. This process is manual: if we decide a certain mutation is not useful and that the whole class of mutants should not be created, the rule is added to the *expert* function. This is the critical part of the system because, without it, users would become frustrated with non-actionable feedback and opt out of the system altogether. Targeted mutation and careful finding surfacing has been critical for the adoption of the mutation testing in Google.

| | Name | Scope |
|---|---|---|
| AOR | Arithmetic operator replacement | a + b $\rightarrow \{a, b, a-b, a*b, a/b, a\%b\}$ |
| LCR | Logical connector replacement | a && b $\rightarrow \{a, b, a\|\|b, true, false\}$ |
| ROR | Relational operator replacement | a >b $\rightarrow \{a<b, a<=b, a>=b, true, false\}$ |
| UOI | Unary operator insertion | a $\rightarrow \{a++, a--\}$; b $\rightarrow$ !b |
| SBR | Statement block removal | stmt $\rightarrow \emptyset$ |

**Figure 3: Mutation operators**

Heuristics for arid node detection used at Google for each language are described in Appendix A.

# 5 ANALYSIS OF DIFF-BASED PROBABILISTIC MUTATION ANALYSIS RESULTS

The dataset contains information on 1'159'723 mutants in seven programming languages: C++, Java, Python, Go, JavaScript, TypeScript and Common Lisp. The distribution of mutations is provided in Figure 4. In total, 72'425 diffs were analyzed as part of the code review process and 150'854 actionable results generated, out of which 11'049 got developer feedback. The purpose of the analysis is to better understand the efficacy and percieved usefulness of mutation types accross programming languages.

## 5.1 Dataset

The analyzed dataset contains information about all the mutation analyses that were run. For each diff analyzed, the dataset contains:

- files with lines affected,
- 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 context and mutator types for each mutant.

| Language | Count (Ratio) | Survival rate |
|---|---|---|
| Java | 543'541 (47%) | 13.2% |
| C++ | 279'575 (24%) | 11.7% |
| Python | 129'868 (11%) | 14.7% |
| Go | 1'050.7 (9%) | 14.0% |
| JavaScript | 86'123 (7%) | 13.1% |
| TypeScript | 13'318 (1%) | 8.3% |
| Common Lisp | 2'272 (1%) | 1.0% |

**Figure 4: Mutants per programming language**

## 5.2 Results

We first investigate the mutation operator survivability (Section 5.3). Then, in Section 5.4 we investigate the developer feedback on the surfaced mutants. Data is sliced by language and mutator operator types for deeper insight. Programming languages have different characteristics, so it is reasonable to expect the mutation analysis results to vary correspondingly. Out of 1'159'723 mutants in the dataset, the majority of mutants come from Java (47%) and C++ (23%), as shown in Figure 4.

| Mutation operator | Count (Ratio) |
|---|---|
| SBR | 829'999 (72.18%) |
| UOI | 197'776 (17.19%) |
| ROR | 51'574 (4.48%) |
| LCR | 51'360 (4.46%) |
| AOR | 19'448 (1.69%) |

**Figure 5: Mutants per mutation operator**

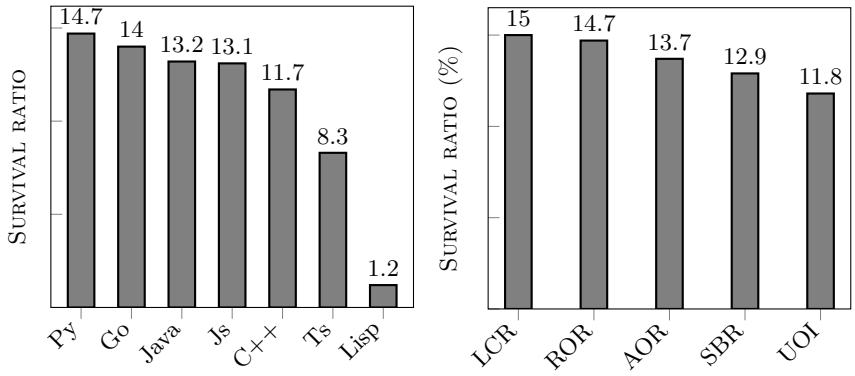
## 5.3 Mutation survivability

Over 87% of all test runs over mutants fail, killing the mutant. This is not the mutation score [4] (the ratio of mutants killed to total number of mutants) because of the probabilistic nature of mutagenesis where only a subset of mutants is generated and evaluated, and many potential mutants are not ever tested because they are in arid nodes.

It is interesting to look at the different breakdowns of the mutants by various criteria. Differences between programming languages are expected. Indeed, Python mutants have the highest survival rate of 14.7%, shown on Figure 4. We argue that the difference is related to the fact that C++ is a compiled language and a whole class of faults that appear in Python code are caught by the C++ compiler. Similarly, TypeScript is optionally statically typed superset of JavaScript and shows lower survivability. It is interesting that Go is second with 14%.

Figure 6 shows the survivability of each type of mutant. Because SBR mutation can be applied to almost any non-arid node in the code, it is no surprise that it dominates the mutant space, contributing roughly 72% of all mutants. SBR is a powerful weapon, but a blunt one: it is the mutation type second-least likely to survive the test suite, and is thus surfaced during code review with the probability of 12.9%. However, the distribution of survival probabilities over mutation types is quite stable, with LCR (Logical Connector Replacement) being the most robust at 15%.

Different programming languages exhibit different behaviors, so it is worth looking into survival rates per language. Figure 12 shows the overall distribution, with the TypeScript AOR mutator as the best fit mutator with a 22.7% survival




**(a) Survival ratio per language** **(b) Survival ratio per type**

**Figure 6: Mutant survival rate sliced by type and language**

rate. Interestingly, Java mutator survivability distribution has the highest entropy, where all operators fall within a narrow 2.6% bucket, while TypeScript has the widest range from 4.9% to 22.7%. Looking at the UOI mutations, it is peculiar that they are the most surviving in Python while in all other languages they are the least surviving mutants. We argue this is due to extensive use of boolean literals in Python's default parameter values that are not tested or used for injection of testing doubles.

When generating only a single mutant in a line, survivability data can be leveraged to predict the best mutant to generate in that line, based on historic data.

### 5.4 Developer feedback

User feedback is gathered via Critique (Section 2.3.1) where each surfaced code finding displays "Please fix" and "Not useful" links. 75% of all findings with feedback were found useful by developers. This ratio gets better over time because the "Not useful" feedback can usually be generalized in a rule for the *expert* function described in Section 4, so future mutations of nodes in which mutants were found not to be useful can be suppressed, generating fewer useless mutants over time. Some of the heuristics of the *expert* for several languages are described in Appendix A.

Living mutants are a precondition for surfacing an actionable finding, but alone do not make a good measure of efficacy. A naïve example would be a mutation that changes the deadline of a network call; this mutant would likely survive, but rarely be useful and effective. Developer feedback, requesting the author of the diff to improve the tests based on the mutant (by means of "Please fix") is a stronger signal that the mutant is effective. In Figure 7, we present the perceived usefulness of mutants by the operator type, LCR being useful in most cases (87.3%) and AOR the least (61.7%).

Another angle is to view the perceived usefulness sliced by programming language, shown in Figure 7. It's interesting to note that findings in JavaScript and Go have lower value that in other languages. We argue that in JavaScript, this is due to mutants in nodes associated with type annotations and module imports. For Go, it stands to reason that its idiomatic way of handling errors from calls that is omnipresent but rarely exhaustively tested is the cause. These point to areas of improvement in the arid node detection mechanism.

We find that developer feedback is the most important available measure for mutation analysis results due to Google's high accuracy and actionability requirements for surfacing findings during code reviews.

## 6 FUTURE WORK

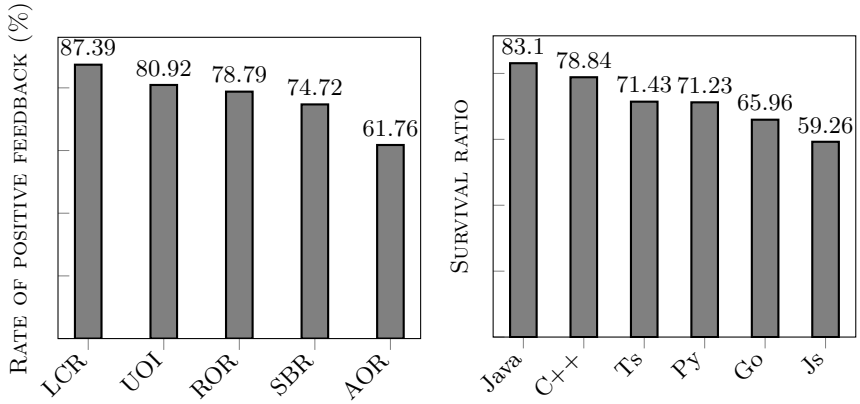
Because at most one mutant is generated per line, it is important to maximize the probability that the generated mutant survives and is actually useful to the developer. Since developer feedback and mutant survival are tracked, we will work on a learning system for predicting the usefulness of mutant operator types when faced with multiple available choices. We are working on describing the mutation context by looking into the neighboring nodes in the AST along with other features of the mutation. The usefulness prediction of different mutator types is based on the usefulness of the previous mutations that occurred in similar contexts, i.e. its AST neighbors in a lower-dimension space.

While we have reduced the non-actionable result rates to 25% manually, this approach does not scale since changes have to be implemented for all languages, and when adding support for new ones, all applicable heuristics need to be ported to it.

Our future efforts are geared towards two results: reducing the ratio of non-actionable mutation analysis results, and inferring the arid nodes directly from developer feedback without manual curation.

## 7 CONCLUSION

Developer attention is a valuable resource and should be used with care. In large systems, we have found there to be too many living mutants to efficiently find and surface the surviving ones in a useful and actionable manner. The




**(a) Usefulness by mutation operator** **(b) Usefulness ratio per language**

**Figure 7: Mutant usefulness ratio per language and type**

proposed probabilistic diff-based approach with arid node detection significantly lowers the computation cost and cognitive overhead of surfacing mutation analysis results. The proposed approach surfaces surviving mutants in interesting lines relevant to the author and the reviewers, pointing to potentially flawed or missing test cases within minutes of the code review request. We found 75% of the surfaced findings with feedback to be reported useful, and we have observed many cases in which mutants caught actual bugs in the code, and even more where test suites were improved to kill the reported mutants.

## REFERENCES


[1] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[2] Go Authors. 2016. Go Slice Types. https://golang.org/ref/spec#Slice_types. (2016).

[3] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41.

[5] Google Inc. 2006. gRPC: A high performance, open-source universal RPC framework. https://grpc.io. (2006).

[6] Google Inc. 2006. Guice: a lightweight dependency injection framework for Java 6 and above. https://github.com/google/guice. (2006).

[7] Google Inc. 2008. gflags: C++ library for commandline flags processing. https://github.com/gflags/gflags. (2008).

[8] Google Inc. 2009. The Closure Compiler: a tool for making JavaScript download and run faster. https://developers.google.com/closure/compiler/. (2009).

[9] Google Inc. 2015. Bazel build system. https://bazel.io/. (2015).

[10] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Software Conference (ICSE), 2013*. 672–681.

[11] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 654–665.

[12] Steven S Muchnick. 1997. *Advanced compiler design implementation*. Morgan Kaufmann.

[13] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996), 99–118.

[14] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An experimental evaluation of selective mutation. In *Software Conference (ICSE), 1993*. 100–107.

[15] A Jefferson Offutt and Jeffrey M Voas. 1996. Subsumption of condition coverage techniques by mutation testing. (1996).

[16] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59 (2016), 78–87. http://dl.acm.org/citation.cfm?id=2854146

[17] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.

[18] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Software Confenrece (ICSE), 2015*.


## A ARID NODE EXPERT RULES

The *expert* function defined in Section 4 consists of various rules in Google, some of which are mutation-type-specific, and some of which are universal. The rules are defined at the AST node level.

### A.1 General rules

There are a few rules that are applied for all languages and mutation types.

Program arguments are commonly passed to Google binaries using the gflags library [7]. Flag definitions declare the default value of the flag, and a test suite will rarely check for changes in this behavior. Empirically, mutations in flag definitions have been found predominantly useless, so the flag definition nodes are considered arid nodes by the *expert*.

```
DEFINE_bool(dry_run, true, "debug only");
```

```
DEFINE_bool(dry_run, false, "debug only");
```

The most common arid nodes are logging statements, which are never tested outside of the logging framework itself. Logging nodes are considered arid nodes by the *expert*.

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

Import statements are not interesting nodes, and removing them often causes compiler or interpreter errors, so they are considered arid nodes by the *expert*.

Return statements are not mutated either. Note that the only mutation type applicable to the return statement is SBR, which would delete it and likely cause a compiler error. The expression contained within the return statement itself is a node of another type and is subject to mutation. For example, the following return statement will never be deleted, but the division operator might be mutated: *return* $f(3 \ / \ 4f)$. The return statement in the AST given by Figure 8 will not be mutated, but its children might.

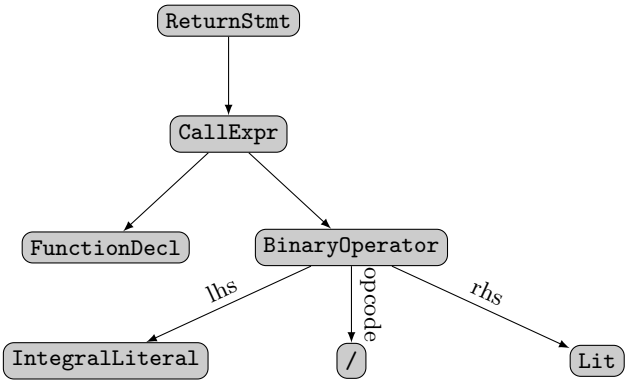



**Figure 8: Possible AST for a return statement**

Time specification-changing mutations are avoided because unit tests rarely test for timing conditions, 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 [5] server's Wait function that is always invoked in RPC servers, which are abundant in Google's code base) are considered arid nodes by the *expert* function.

```
sleep(100); rpc.set_deadline(10);
```
```
sleep(200); rpc.set_deadline(20);
```

Memory-reserving functionality is not mutated because it usually leads to slower but equivalent mutants. A common example is free, delete or `std::vector::reserve`, that potentially causes a reallocation and sets the vector capacity, but exhibits no semantic difference; deleting this line or changing the allocation size might cause more reallocations on the way or a segmentation fault, but will probably not cause a failure that should be caught by a test. Such nodes across languages are considered arid by the *expert* function.

Memoization is often used to improve code execution speeds, but its removal inevitably causes the generation of an equivalent mutant. An equivalent mutant is a program that is syntactically different from the original, but semantically equivalent to it. The question of equivalence is unfortunately undecidable, so avoiding generating equivalent mutants is important. 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. An example cache-lookup in Go is shown in Figure 9. Removing the emphasized block would just kill the cache, but the program would still work in the same way, thus the change is not detectable by any semantic tests.

```
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
}
```

**Figure 9: Memoization in Go**

In the for-statement condition, the less than operator is not mutated to a not-equal operator. This usually results in the equivalent mutant and is suppressed. An example is given in Figure 10. Replacement with other operators is not suppressed.

```
for (int i = 0; i < 10; i++)
```
```
for (int i = 0; i != 10; i++)
```

**Figure 10: Equivalent mutant in `for` loop condition**

*if* statements that contain only arid nodes are arid nodes. If the *if* statement's *then* and *else* (and other) blocks contain nothing but arid nodes, they and their conditions are not mutated. The same is true for all expressions containing one or more blocks as children, e.g. *for*, *while*, *do*, *case*, *switch*, etc.

```
if (FLAGS_debug) {
  LOG(INFO) << "request received: "
            << req->DebugString();
}
```

```
if (!(FLAGS_debug)) {
  LOG(INFO) << "request received: "
            << req->DebugString();
}
```

## A.2 Arid node expert rules specific to a certain programming language

The heuristics used for different programming languages are highly driven by the internal frameworks and constructs and they evolve over time. A small window into them is given in the following language-specific subsections.

*A.2.1 C++.* The LCR mutator has an exception when dealing with `NULL` i.e., `nullptr`, because of its equivalence with *false*. The example in Figure 11 demonstrates this. The mutants marked in bold are equivalent because of the falsy value of `nullptr`. The same is true of the opposite example, where the condition is `if (nullptr == x)`, and the left-hand side is equivalent to the mutant where the expression is replaced by false. These mutation subtypes are suppressed.

| ORIGINAL NODE | POTENTIAL MUTANTS |
|---|---|
| $if(x != nullptr)$ | if (x)<br>**if (nullptr)**<br>if (x == nullptr)<br>**if (false)**<br>if (true) |

**Figure 11: Equivalent potential mutants for LCR**

Label statements and declarations are not mutated because of prevalent compiler errors caused by the only mutator applicable in most cases: SBR. Binary operators are affected by SBR only if they are assignment operators, because otherwise they mostly cause compilation errors, e.g., removing the condition in a for statement. The condition itself, as a binary operator, is subject to most other mutations, though. If blocks are removed, they are replaced with an empty block , to avoid compilation errors. Block statements are enclosed by curly braces and are usually children of nodes with blocks, like if, for or while statements. Both the if statement and its block can be fully removed by SBR.

*A.2.2 Java.* Guice-related nodes [6] are considered arid nodes by the *expert* function (see section 4). Java's java.lang.Object methods that are usually generated, namely equals, hashCode, clone, toString are considered arid nodes by the *expert* function. Method, constructor, enum, interface, variable and class declarations are considered arid nodes by the *expert* function. SBR, as the only applicable mutation type, would usually cause compilation errors if applied. Exception mutation is mostly suppressed. Deleting throw, throws and catch statements can cause compilation errors, and mutating the exception messages is rarely asserted upon; tests usually check the type of the thrown exception (e.g. `PermissionException`). Such nodes are considered arid by the *expert* function and are not mutated.

*A.2.3 Go.* Go has a relatively simple grammar which results in fewer rules for the *expert*. Go's `make` built-in function creates a slice of certain length, and potentially capacity. Additionally, runtime.KeepAlive is marked as arid, because it deals with scheduling finalizers and freeing objects, and is never tested by unit tests. Mutants in variable declarations are suppressed. Go is very special in the way it handles unused variables and imports: they are compiler errors and that behavior cannot be changed by compiler flags, unlike gcc or clang `Wno-error` flag family. Deleting the only statement using a variable in Go will result in a compilation error in the line declaring the variable. Deleting the only statement using a module, like `math.Pow`, will result in the compilation error in the line importing the math package. Go also has a shorthand for variable declaration, e.g., `k := 3`. All such nodes are considered arid nodes by the *expert* function and mutants in them are suppressed.

Go also has a very minimalistic approach to its core libraries and the language in general. There are no size or empty functions, and a built-in len function is used instead. It is common to check the emptiness of the slice by asking whether `len(slice) > 0`, and a mutant reversing that operator to less than produces unreachable code, because by definition, the length of a slice is non-negative. ROR mutations of conditions comparing length of a structure and 0 are limited in a way that the mutation of unreachable code is avoided, covering both sides of the mirrored comparisons, where 0 is on the left- or right-hand side.

Go slice [2] capacity specification in make statements are arid nodes, and mutating them produces equivalent mutants.

```
s := make([]string, len(x) + 2)
```

```
s := make([]string, len(x) * 2)
```

*A.2.4 Javascript.* The Google's Closure library [8] is commonly used in the codebase. Its namespace features for handling packages and avoiding name collisions are prevalent and are considered arid nodes by the *expert* function. Some of those nodes include calls to goog.require, goog.provide, goog.module, module.register.

JavaScript has deep and shallow equality operators (==, ===, !=, !==). There are heated discussions on how these should be used, but it is generally not useful to mutate between deep and shallow variants of the same operator, especially since the opposite operator mutation usually makes

for a good mutant. LCR mutations are limited to the opposite operator, i.e. == → != and === → !==.

Closure compiler type hints are considered arid nodes by the *expert* function: removing them does not and should not break any tests. Various constructor and constructor-like field definitions in JavaScript are considered arid nodes.

*A.2.5 Python.* When python is running a module as a main program, it sets the special *__name__* variable to "*__main__*". Most python binaries contain the following boilerplate.

```
if __name__ == "__main__":
  run()
```

This *if* statement is considered an arid node by the *expert* function. Inverting the condition does not produce a useful mutant, as the condition will only evaluate to true when the python file is executed as a stand-alone program, not when executed within a test.

Python's print and assert builtin functions are considered arid nodes by the *expert* function and are not mutated.

Default argument values are considered arid nodes by the *expert* function. Based on a lot of feedback from developers, these mutations were suppressed because of their low value. Example of such mutant is demonstrated in the following listing.

```
def f(size, duration, annotate=False):
```

---

```
def f(size, duration, annotate=True):
```

While this might suppress important mutants, the overwhelming feedback of low usefulness prevailed in marking these nodes arid. This might come from the Google-specific Python style and tests.

| Programming Language | Survival rate (%) |
|---|---|
| TypeScript AOR | 22.727 |
| C++ LCR | 18.497 |
| Python UOI | 17.915 |
| JavaScript LCR | 17.868 |
| Go ROR | 17.860 |
| C++ AOR | 16.673 |
| JavaScript ROR | 15.297 |
| Go SBR | 15.029 |
| Java AOR | 14.975 |
| Go AOR | 14.964 |
| C++ ROR | 14.535 |
| Java LCR | 14.224 |
| Java UOI | 14.216 |
| Java ROR | 13.763 |
| JavaScript SBR | 13.340 |
| Python SBR | 13.168 |
| Python LCR | 13.018 |
| C++ SBR | 12.644 |
| Java SBR | 12.608 |
| Go LCR | 12.042 |
| Python ROR | 10.979 |
| JavaScript AOR | 10.776 |
| Python AOR | 8.421 |
| TypeScript SBR | 8.408 |
| TypeScript ROR | 8.092 |
| Go UOI | 7.785 |
| TypeScript LCR | 7.083 |
| JavaScript UOI | 6.608 |
| C++ UOI | 5.178 |
| TypeScript UOI | 4.861 |
| Lisp SBR | 1.757 |
| Lisp LCR | 1.666 |
| Lisp UOI | 0.740 |

**Figure 12: Survival rate per language and type**