Mutation testing at Google

Omar Aguirre Rodríguez, Raúl Antuña Suárez, Samuel de la Calle Fernández, Carlos Sampedro Menéndez, Pablo Rodríguez García

Table of Contents

What is mutation testing

Mutation testing is a testing method that allows you to assess your test suite's efficacy. To put it simply, it's a way of testing your tests. It works by introducing small faults into your code and checking if your tests detect them. These modifications are called mutants.

Why mutation testing

The reason for doing this is that when coding we are naturally going to introduce many bugs. We will probably easily find and fix many of them, but we are most likely going to miss some of them.

So, we want to introduce some bugs on purpose to check if our test suite is able to detect them, to make sure that our test suite will also be able to detect any actual bugs we might introduce accidentally in the future

What are mutants

Mutants could be things like changing and equals for a not equals, a plus for a minus, an and for an or.

These might seem like very stupid bugs that we will not introduce naturally, but they could correlate to real bugs.

Even if you might not accidentally change a + for a -, you might introduce some bug in some step of calculating some value, maybe even in a component that provides a value used in that calculation, and the final result ends up being wrong, so you want to see if your test suite will detect a wrong result in that operation, and for that, changing + to – is enough.

Mutation testing metrics

To evaluate our tests efficacy, we use the mutation score, which is the percentage of the mutants that were killed. When we say killing a mutant, we mean that our tests detected that mutant.

The mutation score isn't as well defined as other typical metrics like line coverage. Line coverage is a really objective metric. Either a line is covered by some test or it's not. If you have 100% line coverage you know for a fact that the entire code is covered by tests.

However, mutation score depends on the quality of the mutants. Maybe you have a 100% mutation score, but you had very few mutants or they were very easy to find, and it's not fair to compare that score to maybe another project that has a 70% but has a lot of really good mutants. So mutation score is much more subjective.

Mutation Testing Process

1. **Run tests on the original program**
   a. The test suite is executed on the original program to obtain its expected results.
2. **Mutant generation**
   b. Mutants are created by applying mutation operators to generate modified versions of the original program.
   c. Each mutant represents a possible fault artificially introduced into the code.
3. **Run tests on the mutants**
   d. The test suite is executed on each mutant, generating their respective test results.
4. **Comparison of results**
   e. The results of the mutants are compared with those of the original program:
      i. If the results are the same, it means the test did not detect the error, and the mutant survived.
      ii. If the results are different, it means the test detected the fault, and the mutant has been "killed."
5. **Test adequacy calculation**
   f. The effectiveness of the test suite is measured by calculating the mutant kill rate.
   g. A low percentage indicates that the test suite is not effective enough to detect errors in the code.

Operators

To perform mutation testing we must apply **mutation operators** to existing pieces of code. These mutations would make possible the **generation of new test cases**. It's possible to **test method invocations** by the parameter values, or swapping the parameters, or even adding new parameters (whenever possible).

Some way of misunderstanding what a mutation test is, are **change detector tests**. These tests just check if some tiny implementation detail changes. For example, an exception message. In mutation testing we are not interested in these kinds of details, since we are **forcing some implementation** and not caring about the purpose of testing.

Mutagenesis (a tool from Google)

To deal with mutation testing at Google, they created a tool called Mutagenesis that is executed after some pull request arrives. Follows these steps:

1. Awaits **coverage calculation** (parts of code that the testing tools managed to reach during their evaluation process)
2. Calculates a map from each covered line to a **set of tests that test this line**. (distinguish those tests that reach that line from those that don't, making it much cheaper than just running all the tests for each mutant)
3. This tool **generates mutants using this mutagenesis process**. (filter productive lines with un-productive lines, with an heuristic process)
4. The remaining lines will be mutated only a single mutation.
5. The set of mutants will be **created and text run executed against them**.

Programming languages

A broad wide range of programming languages are being mutation tested, some of them include C++, Python, Java, Go, TypeScript, JavaScript, Kotlin, Dart, Common Lisp, SQL, and most recently Rust.

Mutation testing strategy

The mutations must have some kind of strategy, to avoid losing the developers attention that is a very important resource. If mutation testing is not implemented with this in mind, we could overload the senses of the developer with too many information or waste time running unnecessary tests. To avoid this the strategy of the mutation goes only apply mutation testing to the changed lines in the pull requests made by the developers. To do this, it must use an AST that allows to make precise modifications to source code for mutation testing.

The role of AST

The AST is used to analyse and modify source code for the mutation testing. Each language has its own AST implementation (e.g. Clang for C++, Java parser for Java). No universal AST can be used due to limitations in type information and language complexity. In the case of Google, the company optimizes mutation testing per language using its native AST parsing capabilities.

Arid nodes

Some code areas are less interesting for mutation testing (e.g. logging, memory reservations). Surfacing mutants in these areas wastes developer time and increases cognitive load. To avoid this from happening the arid nodes are considered as zones of the code where the mutation testing is suppressed.

These nodes are simple nodes (a simple node is a node with no children) or compound nodes whose children are all arid nodes. The idea is to suppress the mutation testing in these zones to let the developers focus on high impact faults.

Scaling mutation testing at Google

In google they have implemented mutation testing in a way that it can be scaled. despite 500M+ tests/day and 60K+ code changes/day, mutation testing remains scalable. To achieve the scalability of mutation testing, google follows the following ideas:

- Mutation testing is not run over very large code changes

- Mutation testing is only run over human generated code

- Have aggressive suppression heuristics, focusing only on relevant mutations.

- Skip large-scale cleanups.

By applying this we obtain a performance that runs test in only a few minutes.

Google's implementation of mutation testing

Google’s implementation is different from open-source mutation testing because:

- A lot of open-source tools mutate bytecode, or similar low-level approaches.
- Google's implementation instead modifies the source code's Abstract Syntax Tree.

The reasoning for this decision is because it allows better visualization for developers since, for example, with bytecode mutation, showing how mutants function would be harder.

Meta mutants

A meta mutant is the result of a technique where you embed all the mutants you have into just one single entity, said entity being the meta mutant.

They have not adopted this approach in Google, because:

- They have very good object caching, which makes the potential improvements not that big of a deal.
- Goran wants to tackle this implementation in the future, but he has not had the time to do so yet.

Mutation testing and its effect on developers

To check the effects this practice would have on developers, Google collected data about mutation testing for a period of 6 years, including about 15M mutants.

The results of this showed that developers are inclined to write more test cases when mutants come into the equation (since they will get warnings about needing to get rid of them). It is important to mention that these tests are not simply lazy ones to increase the coverage with no real asserts, the data collected showed that most actually killed mutants.

Coupling effect

The coupling effect is that mutants don't necessarily look like real bugs do. However, they do operate in a similar fashion, that is, if you develop test cases to catch these mutants, it is very likely that said tests will catch real bugs that are somewhat similar too.

The way to measure the coupling is to see how many bugs correspond to a mutant (if the mutant is killed, so will the bugs).

To evaluate the actual correlation between killing mutants leading to killing real bugs, Google conducted an analysis, with thousands of bugs as a sample.

While it is difficult to get accurate data due to how each project handles and documents bugs, they managed to obtain that in approximately 70% of cases, the bug and the mutant were actually coupled, which means that, had they used mutation testing earlier, the bug wouldn't still be active in the product.

While the actual cost of this analysis was quite expensive, as Goran says, its results showed that mutants did help with bug prevention and that they were worth investing in.

Current challenges

- Computational Expense As noted in the documentation, "The number of possible applicable mutations is extremely high, even for a small piece of software." This creates significant computational overhead as each test case must be re-executed against every mutant. At Google Shopping, this challenge became particularly evident when attempting random mutation approaches, which proved unsustainable despite being initially interesting.
- Equivalent Mutants One of the most persistent challenges is identifying equivalent mutants - those that behave identically to the original code despite being syntactically different. As the speaker states, "It is very difficult to recognize analytically what mutants are equivalent." This problem wastes computational resources and human attention, which was identified as "the most valuable resource" in our real-world implementation.
- Mutant Quality Not all mutants are equally valuable. For example, our experience showed that "all mutants regarding caching are useless as all of them are equivalent." The speaker confirms this issue noting that "some faults lead to syntactic errors and are already revealed by the compiler" while others may not represent realistic defects.

Future Directions

- Intelligent Mutant Selection Rather than generating all possible mutants, the future lies in intelligent selection. The documentation mentions that "often it is decided to apply only samples of mutation operators." Our practical experience confirmed this, as we "ended up with 5 or 6 groups" of useful mutations.

- Heuristic Approaches As the speaker describes, search-based software testing using genetic algorithms and other heuristics offers promising advances. These approaches "lead, more or less quickly, to discover test suites with good overall fitness values." At Google Shopping, we found that "many improvements can be done with heuristics" to discard useless mutants.
- Integration with AI Tools like TestSpark demonstrate how AI-based approaches are being integrated with mutation testing. This represents a significant future direction where mutation generation becomes more targeted and efficient, combining "LLM-based test generation" with "local search-based test generation."
- Conclusion: The future of mutation testing lies not in more comprehensive mutation but in smarter mutation - focusing human attention where it matters most, using heuristics to identify valuable mutants, and

integrating with AI technologies to make the process more efficient and effective.

Practical Implementation Advice

From our experience, effective mutation testing isn't about achieving 100% mutation coverage but improving product quality.
As we discovered, "the goal is not just to kill mutants but to improve the product overall."

The learning curve is steep but rewarding: "At first it is annoying, however the more you do it, the more you learn and improve
towards improving coverage and avoiding redundant mutants."

Our recommendation is not to force "an additional set of mutant tests to your already present tests but to do a little bit of them every day"
and "learn how to suppress mutants to avoid having 100s of them."