Mutation Testing in the Wild: Challenges and Lessons from Bitcoin Core

Bruno Ely Reis Garcia
Vinteum
Universidade de São Paulo
bruno@vinteum.org

Abstract

Mutation testing is a technique for evaluating test suite effectiveness, yet its adoption in industrial settings remains limited due to scalability challenges. In this paper, we report our experience applying mutation testing to Bitcoin Core, the reference implementation of the Bitcoin protocol - a safety-critical, C++ codebase with hundreds of thousands of lines of code and a heterogeneous test suite. We present **bcore-mutation**, an open-source tool written in Rust and tailored to Bitcoin Core's specific constraints, and describe the challenges we faced: test selection for incremental analysis, unproductive mutant filtering, and surfacing results to pull request authors without disrupting the review workflow. We further discuss how mutation testing was applied to evaluate the quality of fuzz harnesses, and report an experimental use of Large Language Models to automatically generate tests that kill surviving mutants. Our experience shows that mutation testing is tractable even on large, complex codebases when the tooling is designed around the realities of the project, and offers lessons for practitioners considering similar approaches on safety-critical systems.

Keywords

Mutation Testing, Software Testing, Open Source

1 Introduction

Software testing is a cornerstone of software quality assurance, yet measuring the *effectiveness* of a test suite remains a hard problem. Code coverage metrics are widely used in practice, but they are known to be poor proxies for test quality: a line can be executed without its behavior being meaningfully verified [3]. Mutation testing addresses this gap by systematically injecting small syntactic faults - *mutants* - into the source code and checking whether the existing tests detect them. A mutant that is not detected by any test, called a *surviving mutant*, signals a potential blind spot in the test suite.

Despite its theoretical appeal, mutation testing has seen limited adoption in industrial settings, largely due to scalability concerns [1]. Generating and evaluating thousands of mutants demands significant computational resources, and the cost multiplies on large codebases with slow build and test cycles. As a result, most practical accounts of mutation testing at scale come from a handful of large technology companies [4].

Bitcoin Core is an especially challenging and consequential subject for mutation testing. As the reference implementation of the Bitcoin protocol, it underpins a financial network securing hundreds of billions of dollars in value. Its codebase is written in C++, spans hundreds of thousands of lines, and encompasses a heterogeneous test suite combining unit, functional, and fuzz tests. A single full build can take tens of minutes, and some functional tests individually exceed one minute of execution time. These characteristics make mutation testing even more expensive.

In this paper, we report our industry experience applying mutation testing to Bitcoin Core. We describe [1]**bcore-mutation**, an open-source tool written in Rust and tailored to Bitcoin Core's specific constraints, and present the challenges we encountered - from test selection and unproductive mutant filtering to incremental analysis at pull-request granularity. We also discuss how mutation testing was extended to evaluate the quality of fuzz harnesses. Our goal is not only to document what we built, but to share lessons that practitioners working on other systems can apply when considering mutation testing in their own contexts.

2 Challenges

In this section, we present the challenges regarding our actual use of mutation testing.

2.1 What Tests Should We Run?

Bitcoin Core has hundreds of unit tests and functional ones, in addition to fuzz testing. Some individual functional tests may take more than one minute to execute, depending on hardware. This means that running all tests for every mutant is impractical. When performing incremental mutation testing, given a mutated file, we need to determine which tests should be run for that mutant. In theory, we could generate a coverage report and identify which tests execute the mutated code. However, this does not work as well as expected. A test may execute a portion of the code without necessarily testing its behavior. So, relying on coverage report may inflate the number of tests that we need to run. This is still an open question for us and we currently still need tons of manual work.

2.2 How to Show Up the Surviving Mutants from a Pull Request?

In order to perform incremental mutation testing, we need to define how to show up the surviving mutants to the pull request's author. We are following same recommendations of Petrović et al. [4] (Google's report) where they mention to report at most 7 times the number of total files from the mutated PR. We are currently approaching two different ways of reporting the surviving mutants. The first one is commenting directly on the GitHub Pull Request and the second is to publish them on the same website where we report the weekly runs. Our concern about commenting directly on Pull Requests is that we do not want to cause any noise. However, it appears to be the most effective way to draw the PR author's attention to the surviving mutants.

[1] https://github.com/brunoerg/bcore-mutation

2.3 How to Avoid Unproductive Mutants?

One of the pains of mutation testing is the equivalent mutant problem. A mutant is called equivalent when its behavior is exactly the same as that of the original program, thus it cannot be killed. There are several proposals on how to avoid generating this kind of mutant as well as how to identify whether a mutant is equivalent or not.

However, besides the equivalent mutants, there are also unproductive mutants. They are not equivalent but it is not worth writing tests to kill them. The Listing 1 shows code for which generating mutants is not worthwhile. The reason is that this code is a debug logging block which is purely observational. We could delete this entire block and the program would behave identically; we would just lose the debug output. Our challenge is to avoid generating mutants for code like this.

Listing 1: Part of coin selection code from Bitcoin Core

```
if (LogAcceptCategory(BCLog::SELECTCOINS, BCLog::Level::
     Debug)) {
    std::string log_message{"Coin␣selection␣best␣subset:␣
         "};
    for (unsigned int i = 0; i < applicable_groups.size()
         ; i++) {
        if (vfBest[i]) {
            log_message += strprintf("%s␣", FormatMoney(
                 applicable_groups[i].m_value));
        }
    }
    LogDebug(BCLog::SELECTCOINS, "%stotal␣%s\n",
         log_message, FormatMoney(nBest));
}
```

3 The Bcore-Mutation Tool and Our Current Approach

bcore-mutation is an open-source mutation testing tool written in Rust and tailored to the specific needs of Bitcoin Core. Unlike general-purpose C++ mutation frameworks, it is designed around the realities of working on a large, security-critical codebase with long build times and a heterogeneous test suite (unit, functional, and fuzz tests). The tool follows a two-phase workflow. The mutate phase applies a curated set of regex- and pattern-based mutation operators to the target source, including arithmetic, relational, logical, boundary (off-by-one), and control-flow operators, as well as a dedicated security-focused operator set intended to evaluate the effectiveness of fuzzing harnesses. Generated mutants are persisted in a SQLite database, allowing runs to be paused, resumed, re-analyzed, or restricted to surviving mutants only. The analyze phase then applies each mutant to the working tree, runs a user-supplied test command (e.g. cmake –build && ./build/test/functional/...), and classifies it as killed or survived, optionally failing the run when a configurable survival threshold is exceeded. Several design choices address the practical cost of mutation testing on a project the size of Bitcoin Core:

- Scope reduction: mutation can be limited to the diff of a specific PR or branch, a line range, or only the lines exercised by a coverage report.
- Arid node filtering: an implementation of Google's AST-based arid-node algorithm (arid(N) = expert(N) for simple nodes, conjunction over children for compound nodes) automatically filters out unproductive mutation targets such as LogPrintf, reserve/resize, timing variables, and Bitcoin Core-specific patterns like "***if (G_FUZZING) return;***". Custom expert rules can be supplied via the *–add-expert-rule* flag.
- One-mutant-per-line mode prioritises operators that are historically hardest to kill, dramatically reducing analysis time on large files.
- Test-only mode restricts mutation to unit and functional test files, enabling meta-evaluation of the test suite itself.

Together, these features make mutation testing tractable on a codebase where a single full build can take tens of minutes, and turn it into a tool that can realistically be applied at PR-review granularity rather than only as an offline, whole-repository batch job.

3.1 Weekly Run and Incremental Mutation Testing

Our current setup consists of running mutation testing every week for some files (at this moment we prioritize the most critical files) based on the master branch, and publishing the results on a web page (corecheckhttps://corecheck.dev/mutation and bitcoincore.spacehttps://bitcoincore.space). This process is fully automated, a job runs every Friday and updates the report with the latest results. However, our incremental mutation testing setup is still manual, it requires a user to trigger the run for a pull request as well as deciding what tests should be executed for each mutated file.

3.2 Using Mutation Testing to Evaluate Fuzz Harnesses

Garcia and Souza [2] did an empirical evaluation of fuzz targets using mutation testing. The authors point out that it is possible to write good targets that can go beyond discovering crash-inducing bugs and detect logical bugs as well. Techniques such as metamorphic relations, when incorporated into a fuzz target, improve its ability to find logical bugs. When a contributor opens a pull request claiming that it is improving a fuzz target, for example, adding more invariants, we may run mutation testing on the fuzzed functions in order to know whether the target has improved or not.

One of the examples is the Pull Request [2]#34026 in which the author added fuzz coverage for several functions from the class *CCoinControl* as well as added more invariants to the harness. These new invariants increased its mutation score in comparison with its current version, as can be seen in Figure 1.

3.3 Using LLM to Write Unit Tests to Kill Mutants

As an experimental approach, we are using large language models (LLMs) to write unit tests to kill surviving mutants. The process is not fully automated, and we have not yet evaluated it using a systematic methodology. For some pull requests, the first author personally prompted an LLM to write a unit test giving the existing test cases as context as well as the mutant. We believe that providing a candidate test case to the pull request author may be more valuable

[2]https://github.com/bitcoin/bitcoin/pull/34026

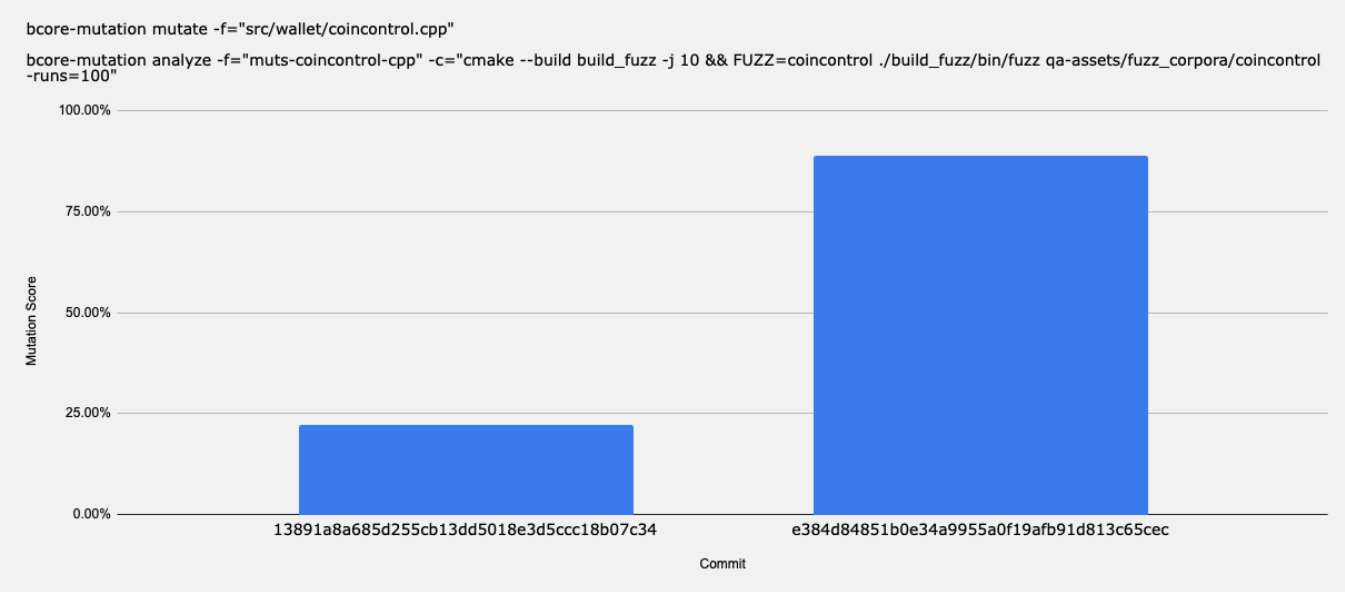

Figure 1: Feedback given to the author of the Pull Request #34026 about his improvements on the fuzz target, showing that his commit substantially improved the mutation score of the fuzzed function.

than providing only the surviving mutant. We noticed that the PR author usually address surviving mutants quickly when we provide a unit test as an example, even the LLM not generating a perfect test case. Refining and improving the generated example is typically less time-consuming than analyzing the mutant and writing a test case from scratch.

4 Conclusion

In this paper, we reported our ongoing experience applying mutation testing to Bitcoin Core, a safety-critical financial system that presents significant challenges for automated test evaluation: long build times, a heterogeneous test suite, and a codebase where correctness has direct economic consequences.

We showed that mutation testing at this scale requires deliberate tooling decisions. **bcore-mutation** addresses the scalability problem through scope reduction, arid node filtering, and one-mutant-per-line analysis, making it feasible to run mutation analysis at pull-request granularity. We also extended the technique beyond traditional source mutation: by applying it to fuzz harnesses, we can give contributors objective feedback on whether their changes meaningfully improve the fuzzing targets.

Several challenges remain open. Test selection for incremental mutation testing still requires significant manual effort, and the question of how to surface surviving mutants to pull request authors without introducing noise in the review process has not been fully resolved. Our early experiments with LLMs to automatically generate tests for surviving mutants are promising, but require further systematic study.

Despite these open questions, our experience suggests that mutation testing can be made practical on large, complex, real-world codebases - provided that the tooling is designed around the specific constraints of the project rather than applied off-the-shelf. We believe the challenges and solutions described here generalize beyond Bitcoin Core, and hope this report is useful to practitioners considering mutation testing for other safety-critical or high-complexity systems.

Artifact Availability

The **bcore-mutation** tool is publicly available at https://github.com/brunoerg/bcore-mutation.

References

[1] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 268–277.

[2] Bruno Garcia and Simone Souza. 2025. An empirical evaluation of fuzz targets using mutation testing. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 75–83. doi:10.5753/sast.2025.14183

[3] K. Jain, G. Kalburgi, C. Le Goues, and A. Groce. 2023. Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Los Alamitos, CA, USA, 102–113. doi:10.1109/ISSRE59848.2023.00036

[4] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3900–3912. doi:10.1109/TSE.2021.3107634