An empirical evaluation of fuzz targets using mutation testing

Bruno E. R. Garcia
ICMC/USP
São Carlos, SP, Brazil
bruno.erg@usp.br

Simone R. S. Souza
ICMC/USP
São Carlos, SP, Brazil
srocio@icmc.usp.br

ABSTRACT

Software testing through fuzzing has gained widespread adoption for discovering security vulnerabilities, yet questions remain about its effectiveness in detecting subtle behavioral faults. This paper presents an empirical evaluation investigating the intersection of fuzzing and mutation testing, specifically examining how well fuzz targets perform when evaluated through mutation analysis. We conducted a systematic study using Bitcoin Core as our subject system, analyzing 10 different fuzz targets across various modules and evaluating their ability to detect 726 generated mutants. Our methodology involved executing fuzz targets with existing seed corpora and measuring mutation scores both with and without assertion statements to understand the role of explicit oracles in fault detection.

Our findings reveal that contrary to previous studies suggesting fuzzing's limited effectiveness in mutation testing, several fuzz targets achieved high mutation scores, with two targets reaching 100% mutant detection rates. We identified three key design patterns that significantly enhance mutant detection capabilities: (1) round-trip testing approaches that verify data integrity through serialization-deserialization cycles, (2) mathematical oracles that implement exact behavioral verification through redundant calculations, and (3) metamorphic relations that validate expected relationships between inputs and outputs. Our analysis demonstrates a positive correlation between assertion density in fuzz targets and mutation scores, with assertion removal causing substantial drops in detection rates across all targets.

The study contributes empirical evidence that well-designed fuzz targets can effectively detect subtle behavioral faults beyond traditional crash-based vulnerabilities. Our results suggest that incorporating explicit oracles, metamorphic properties, and round-trip verification mechanisms into fuzz target design significantly improves their mutation testing performance. These findings have practical implications for improving fuzzing methodologies and developing more comprehensive automated testing strategies for safety-critical software systems.

KEYWORDS

Mutation testing, fuzzing, empirical software engineering

1 Introduction

Software testing remains a critical challenge in ensuring the reliability and security of modern software systems. Among the various testing methodologies, fuzzing has emerged as a particularly effective technique for discovering bugs, especially security vulnerabilities, while mutation testing has established itself as a powerful approach for evaluating test suite quality. This paper presents an empirical evaluation examining the intersection of these two techniques, specifically investigating how well fuzz targets perform when evaluated through the lens of mutation testing.

Mutation testing is a fault-based testing technique that evaluates the quality of test suites by introducing small syntactic changes, called mutants, into the program source code. Each mutant represents a potential fault that mimics common programming errors, such as replacing arithmetic operators, modifying conditional boundaries, or changing variable references [1]. The fundamental principle underlying mutation testing is the "competent programmer hypothesis", which assumes that programmers typically make small mistakes rather than fundamental logical errors. A test suite's effectiveness is measured by its mutation score—the percentage of mutants it successfully detects (kills). A mutant is considered "killed" when at least one test case produces different output between the original program and the mutated version. It has been used in a wide range of different type of systems to improve their testing efforts [7] [11] [6].

Fuzzing, or fuzz testing, is an automated software testing technique that discovers bugs by providing invalid, unexpected, or random data as inputs to a program [12]. Modern fuzzing approaches, particularly coverage-guided greybox fuzzing, have revolutionized vulnerability discovery in complex software systems [2]. Fuzzing operates through several key components: fuzz targets, which are specially crafted entry points that accept fuzzer-generated inputs and exercise specific parts of the codebase; seed corpus, a collection of initial inputs that provides the fuzzer with examples of valid program inputs to mutate and evolve; and coverage feedback mechanisms that guide the fuzzer toward unexplored code paths.

A fuzz target typically consists of a harness function that accepts a byte array from the fuzzer, transforms it into appropriate data structures, and invokes the code under test. The seed corpus serves as the starting point for the fuzzing campaign, containing representative inputs that exercise different program behaviors. The fuzzer then mutates these seeds using various strategies—bit flips, arithmetic operations, dictionary-based substitutions—to generate new test cases that explore previously uncovered code paths. Despite fuzzing's remarkable success in finding bugs, there exists a fundamental mismatch between fuzzing objectives and mutation testing requirements. Fuzzing primarily optimizes for code coverage and crash detection, seeking inputs that exercise new code paths or trigger detectable failures such as memory violations, assertions, or crashes. In contrast, mutation testing requires detecting subtle semantic differences in program behavior that may not manifest as crashes or coverage changes. Many mutants produce changes that are observable only through careful examination of program output values, state changes, or return codes—differences that typical fuzzing campaigns might overlook.

Furthermore, fuzzing's nature and focus on input generation rather than output verification brings challenges for mutation detection [5]. While a fuzzer might generate inputs that execute mutated code, it may not verify whether the program's behavior has actually changed. This is particularly problematic for mutants that modify internal logic without affecting control flow or causing crashes. For instance, a mutant that changes a comparison operator in a sorting algorithm might not crash the program or significantly alter code coverage, yet it fundamentally changes the program's correctness.

This paper investigates these challenges through an empirical study of fuzz targets from a real-world project. We evaluate how effectively fuzzing campaigns can detect various classes of mutants, analyze the characteristics of survived mutants, and identify patterns that explain why certain mutations evade detection. Our findings provide insights into the limitations of current fuzzing approaches when viewed through the mutation testing lens and suggest potential improvements for making fuzz testing more effective at catching subtle behavioral changes in software systems. This study is organized as: Section 2 presents the related work; Section 3 presents all the motivations for this work and the research questions; Section 4 defines the methodology that conducts the study; Section 5 presents the setup to run used to conduct the experiments; Section 6 presents the results; Both 7 and 8 answer the research questions by bringing an analysis and discussion of the results; Section 9 brings some threats to validity; Section 10 concludes the work and, finally, Section **??** brings the link to the artifacts that enable the reproduction of this study.

2 Related work

One of the main references for this work is a paper that investigates how the difference between coverage and mutation score can guide testing efforts [9]. The authors investigate how should a QA engineer spend their time since code coverage might not be a gap on large systems especially because fuzzing can reach high coverage easily. The authors initiate the study with the following question: *"Which cryptocurrency project has better testing practices, Bitcoin Core or Algorand?"*. Then they explore this question by checking test coverage in both projects as well as applying mutation testing.

They point out that Bitcoin Core has a complex, well-designed, set of fuzzing tests that are run on OSS-Fuzz which covers as much code as the project's functional tests. However, they noticed that fuzzing was able to catch just under 12% of all the generated mutants and advocate that the best way to improve this rate is to focus on enhancing the test oracle. Similarly, Groce et al. evaluated the Bitcoin Core's fuzz testing efforts. By applying mutation testing they got the same results as described by Jain et al.. Moreover, fuzz testing eliminating a small number of mutants is not a bad result since it is not practical to write oracles that can deal with a large number of possible inputs [5].

One of the ways of using fuzzing to detect a large number of mutants would be through differential fuzzing [3]. Since writing oracles that can deal with a large number of possible test cases is not practical, it is possible to kill a mutant by comparing the output of the original code and its mutants. Garcia et al. suggest that this approach can be useful to identify equivalent mutants. In same direction, Lee et al. point out that a safety-critical cyber-physical system (CPS) is usually writen in C. However, most tools for testing applications written in C in this context rely on symbolic execution. The authors propose a tool called MOTIF that combines fuzzing with mutation testing. The idea is similar to the proposed by Garcia et al. and their experiments show that it can detect more faults than symbolic execution [10].

Vikram et al. develop and evaluate a Java-based framework for incorporating mutation analysis in the greybox fuzzing loop. The idea is to produce a corpus with a high mutation score. Their implementation relies on a differential oracle where they can kill mutants comparing their outputs with the outputs provided by the original code. From their experiments, their tool was able to produce a better corpus compared to the state-of-the-art java fuzzer Zest [13].

3 Motivation and Research Questions

The effectiveness of fuzzing as a software testing technique has been studied, yet concerns remain about its ability to detect faults that may be present in software systems. Recent studies have highlighted significant limitations in fuzzing's fault detection capabilities, particularly when evaluated using mutation testing as a benchmark [5] [4].

As mentioned in Section 2, fuzzing campaigns often struggle to achieve high mutation scores, failing to kill a substantial proportion of injected mutants. This limitation suggests that while fuzzing excels at discovering crashes and security vulnerabilities through random input generation, it may be less effective at detecting subtle logical errors or boundary condition violations that mutation testing is designed to reveal. The observed gap between fuzzing's practical success in finding real-world bugs and its performance on mutation testing benchmarks raises important questions about the comprehensiveness of fuzz testing and whether current fuzzing approaches are optimally designed for broad fault detection.

Furthermore, the design and implementation of fuzz targets—the entry points and test harnesses that interface between the fuzzer and the system under test—may significantly influence the effectiveness of mutation detection. However, the relationship between fuzz target design choices and mutation killing capability remains underexplored in the literature. Understanding this relationship is crucial for improving fuzzing methodologies and developing more effective testing strategies.

Given these observations, this study seeks to provide a comprehensive empirical evaluation of fuzzing's mutation detection capabilities and investigate how fuzz target design impacts testing effectiveness. Specifically, we address the following research questions:

- **RQ1** - To what extent can fuzz testing campaigns detect mutants?
- **RQ2** - How does the design of fuzz targets impact their ability to kill mutants?

4 Methodology

Our methodology conducts a systematic evaluation of the effectiveness of fuzz targets in detecting mutants, using a controlled and reproducible experimental setup. The process begins with the generation of mutants applied to specific target functions within the codebase. Each mutant introduces a small syntactic change

intended to simulate a potential defect. Following this, we execute the associated fuzz targets —those designed to exercise the mutated functions — using only the existing seed corpus provided by the projects as input. This initial step allows us to compute a baseline mutation score, reflecting the ability of the curated corpus to trigger observable behavioral deviations. These seed corpora, often refined over years of fuzzing efforts, are expected to represent a high-quality set of inputs.

In the second phase of the experiment, we modify the fuzz targets by removing all assertion statements, which are commonly used as lightweight oracles to signal incorrect program behavior. We then re-execute the same fuzz targets with the unchanged corpus and compute a new mutation score. This alteration is critical because assertions typically detect and halt execution upon encountering invalid states, such as out-of-bound accesses or invariant violations. By eliminating them, we can isolate the extent to which mutation detection relies on such explicit checks.

Comparing the mutation scores from both phases enables us to quantify the contribution of assertions to mutant detection. A significant drop in detection rate after removing assertions would suggest that fuzz targets primarily depend on crash-based or assertion-triggered failures rather than capturing more subtle semantic deviations. This comparison sheds light on an important limitation of current fuzzing practices: their tendency to emphasize crash detection over comprehensive behavioral validation. The complete workflow of our methodology is illustrated in Figure 2.

5 Setup

For our empirical evaluation, we selected Bitcoin Core as our primary subject system due to its combination of real-world significance and strong fuzzing adoption. As one of the most critical cryptocurrency implementations, Bitcoin Core represents a compelling real-world system where software reliability is really important. The project has extensively adopted fuzz testing, integrating it as a fundamental component of its quality assurance pipeline. Notably, Bitcoin Core maintains a seed corpus that has been continuously improved over several years, with inputs carefully curated to achieve high coverage across the tested functions. This well-maintained corpus, combined with the project's diverse set of fuzz targets covering critical components like transaction validation, script interpretation, and network protocol handling, provides an ideal foundation for evaluating how effectively fuzzing campaigns can detect mutations in the source code. Furthermore, the project's open-source nature and detailed documentation of its fuzzing infrastructure enable reproducible experiments while ensuring our findings have practical relevance for a widely-deployed, security-critical system.

Given our focus on Bitcoin Core as the subject system for this empirical evaluation, we selected *mutation-core* [1] as our mutation testing for this experiment, since it is a tool designed specifically for Bitcoin Core. This tool has been actively adopted by Bitcoin Core contributors as part of their development workflow, avoids creating unuseful mutants according to the nuances of the project and address most mutation operators that other mutation testing tools for C++ also do. The corecheck project [2] uses mutation-core to generate mutation testing report for Bitcoin Core which is updated every week.

We chose 10 different fuzz targets from different modules, that is, targets that address functions from different areas of the codebase, and we generated the mutants according to the functions addressed by them. Note that some targets might fuzz more than one function. In some cases, we mutated just one function per target and other cases we mutated every function addressed by the target.

Figure 1 shows one of the fuzz targets we chose for this experiment, the Bitcoin Core's script_descriptor_cache. As can be seen, this target addresses more than a single function, contains some assertions as well as ignores the return from some functions - which is common in fuzzing.

```
FUZZ_TARGET(script_descriptor_cache)
{
    FuzzedDataProvider fuzzed_data_provider(buffer.data(), buffer.
        ↪ size());
    DescriptorCache descriptor_cache;
    LIMITED_WHILE(fuzzed_data_provider.ConsumeBool(), 10000) {
        const std::vector<uint8_t> code = fuzzed_data_provider.
            ↪ ConsumeBytes<uint8_t>(BIP32_EXTKEY_SIZE);
        if (code.size() == BIP32_EXTKEY_SIZE) {
            CExtPubKey xpub;
            xpub.Decode(code.data());
            const uint32_t key_exp_pos = fuzzed_data_provider.
                ↪ ConsumeIntegral<uint32_t>();
            CExtPubKey xpub_fetched;
            if (fuzzed_data_provider.ConsumeBool()) {
                (void)descriptor_cache.GetCachedParentExtPubKey(
                    ↪ key_exp_pos, xpub_fetched);
                descriptor_cache.CacheParentExtPubKey(key_exp_pos,
                    ↪ xpub);
                assert(descriptor_cache.GetCachedParentExtPubKey(
                    ↪ key_exp_pos, xpub_fetched));
            } else {
                const uint32_t der_index = fuzzed_data_provider.
                    ↪ ConsumeIntegral<uint32_t>();
                (void)descriptor_cache.GetCachedDerivedExtPubKey(
                    ↪ key_exp_pos, der_index, xpub_fetched);
                descriptor_cache.CacheDerivedExtPubKey(key_exp_pos,
                    ↪ der_index, xpub);
                assert(descriptor_cache.GetCachedDerivedExtPubKey(
                    ↪ key_exp_pos, der_index, xpub_fetched));
            }
            assert(xpub == xpub_fetched);
        }
        (void)descriptor_cache.GetCachedParentExtPubKeys();
        (void)descriptor_cache.GetCachedDerivedExtPubKeys();
    }
}
```

Figure 1: Bitcoin Core's script_descriptor_cache fuzz target

6 Results

Table 1 presents the number of generated mutants for each fuzzed function according to their respective fuzz targets, the assertion density per line of code, and the mutation scores both with and without assertions. Despite prior studies suggesting that fuzzing is ineffective at killing mutants, our results demonstrate that several targets achieved high mutation scores. Furthermore, we observe a positive correlation between the number of assertions in fuzz targets and their corresponding mutation scores, as illustrated in Figure 5.

[1] https://github.com/brunoerg/mutation-core
[2] https://corecheck.dev

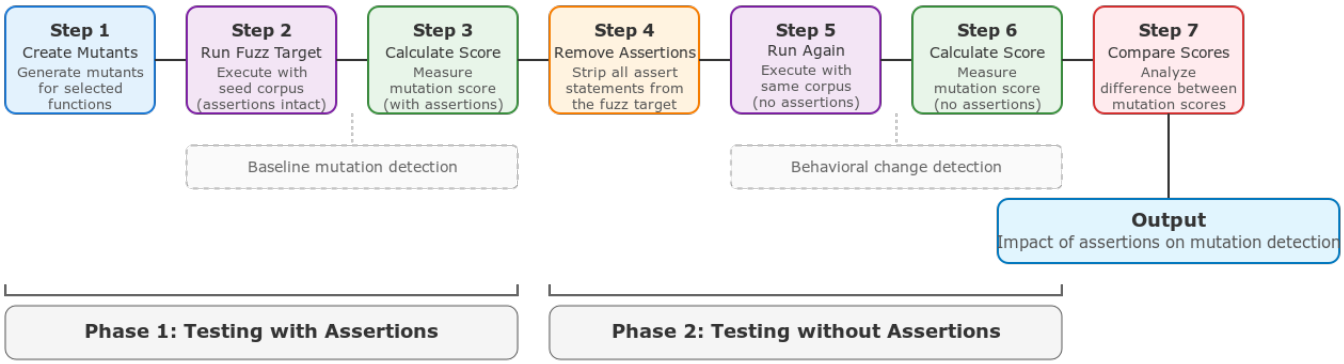

Figure 2: Methodology

Figure 3 compares the mutation scores for each target with and without assertions. Consistent with our hypothesis, removing assertions resulted in a significant decrease in mutation scores across targets. Most notably, the feefrac_div_fallback target experienced a complete drop from 100% to 0% mutation score when all assertions were removed.

Our analysis reveals several key findings:

- **Weak positive correlation**: There's a slight tendency for higher assertion density to correlate with better mutation scores.
- **Notable outliers**: Some targets like feefrac_div_fallback and addrman_serdeser achieve 100% mutation score with different assertion densities.
- **Zero assertions**: Targets with no assertions (net_permissions, parse_numbers) tend to have lower mutation scores.
- **High performers**: bech32_roundtrip has the highest assertion density (0.21) and achieves 82.35% mutation score.

7 RQ1 - To what extent can fuzz testing campaigns detect mutants?

Our experimental results revealed two scenarios where fuzz testing achieved 100% mutant detection rates. Additionally, several cases maintained mutation scores above 80%. Notably, one target that achieved complete mutant detection (100% mutation score) contained approximately 200 mutants — the largest set in our test suite. We can expect fuzzing not to kill a great number of mutants, we show that it is possible to have fuzz targets reaching a high mutation score.

7.1 Round-trip functions

Round-trip functions are a fundamental concept in software testing that involve paired operations where one function performs a transformation on data and its counterpart reverses that transformation, ideally restoring the original input. Common examples include serialization and deserialization pairs (such as JSON.stringify() and JSON.parse()), encoding and decoding operations (like Base64 encoding/decoding or URL encoding/decoding), compression and decompression algorithms (ZIP/unzip), and cryptographic operations with their corresponding decryption functions. The symmetry of round-trip functions makes them excellent candidates for property-based testing and fuzzing, as any deviation from the expected round-trip property immediately indicates a bug in either the forward or reverse transformation, providing a robust oracle for automated testing frameworks.

That said, one fuzz target that achieved a 100% mutation score focused on serialization and deserialization processes, as shown in Figure 4. Any mutation in the serialization or deserialization logic would cause the assertion to fail. This target specifically verifies that data remains unchanged after a complete serialize-deserialize cycle. However, it is important to highlight that, in this case, the serialization and deserialization functions have internal verifications to check if the produced result is correct, it explains why the target reached 100% of mutation score without the assertions in the fuzz target.

Another round-trip case is the bech32_roundtrip fuzz target. It reached a high mutation score of 82.35%, however, removing the assertions and futhermore removing the round-trip verifications, made the mutation score to drop to 44.12%. Bech32 is a Bitcoin address format introduced in BIP 173 (Bitcoin Improvement Proposal 173) that provides a more efficient and user-friendly way to encode native SegWit addresses. Unlike legacy Bitcoin addresses that use Base58 encoding, Bech32 uses a custom base-32 encoding scheme that includes only lowercase letters and numbers, making addresses easier to read, type, and less prone to errors through its built-in checksum mechanism. Bech32 addresses are identifiable by their "bc1" prefix for mainnet transactions and offer several advantages including better error detection capabilities, case-insensitive encoding that reduces transcription errors, and more efficient QR code generation due to the character set optimization [14].

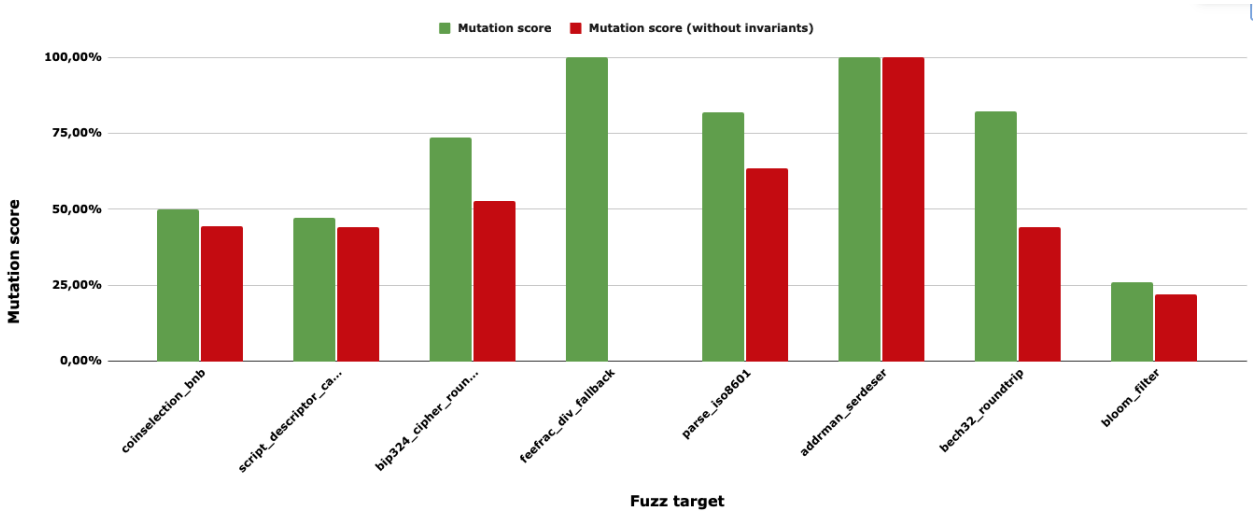

Figure 3: Mutation score with and without any assertion in the fuzz target

Fuzz Target	Mutated Functions	# Mutants	Assertions per fuzz target's LOC	Mutation Score (with assertions)	Mutation Score (no assertions)
coinselection_bnb	SelectCoinsBnB	97	0.093	49.98%	44.33%
net_permissions	TryParsePermissionFlags	78	0	25.64%	N/A
script_descriptor_cache	GetCachedParentExtPubKey; CacheDerivedExtPubKey	34	0.125	47.06%	44.12%
bip324_cipher_roundtrip	Decrypt; Initialize	19	0.11	73.68%	52.63%
feefrac_div_fallback	DivFallback	24	0.159	100%	0%
parse_iso8601	FormatISO8601DateTime	11	0.09	81.82%	63.64%
addrman_serdeser	Serialize; Unserialize	207	0.07	100%	100%
bech32_roundtrip	Encode; Decode	68	0.21	82.35%	44.12%
bloom_filter	insert; contains	146	0.05	26.03%	21.92%
parse_numbers	ParseMoney	42	0	19.05%	N/A

Table 1: Results of evaluating mutation testing with fuzz targets

```
// Check that serialize followed by unserialize produces the same addrman.
FUZZ_TARGET(addrman_serdeser, .init = initialize_addrman)
{
    SeedRandomStateForTest(SeedRand::ZEROS);
    FuzzedDataProvider fuzzed_data_provider(buffer.data(), buffer.size());
    SetMockTime(ConsumeTime(fuzzed_data_provider));

    NetGroupManager netgroupman{ConsumeNetGroupManager(fuzzed_data_provider)};
    AddrManDeterministic addr_man1{netgroupman, fuzzed_data_provider, GetCheckRatio()};
    AddrManDeterministic addr_man2{netgroupman, fuzzed_data_provider, GetCheckRatio()};

    DataStream data_stream{};

    FillAddrman(addr_man1, fuzzed_data_provider);
    data_stream << addr_man1;
    data_stream >> addr_man2;
    assert(addr_man1 == addr_man2);
}
```

Figure 4: Bitcoin Core's addrman_serdeser fuzz target

7.2 Testing the exact behavior

The other fuzz target that killed 100% of the mutants was the feefrac_div_fallback and can be seen in Figure 6. Its performance stems from its implementation of a mathematical oracle with exact expected behavior, which fundamentally differs from typical fuzzing scenarios where determining correct output for arbitrary inputs remains challenging. Rather than relying on external specifications or subjective correctness criteria, this target implements a redundant reference calculation using alternative arithmetic libraries (arith_uint256) that independently computes the mathematically precise expected result for any given input combination of numerator, denominator, and rounding mode. This approach creates what mutation testing researchers term a "strong oracle" [8] — a verification mechanism that can definitively determine correctness for the entire input domain through dual verification methods: exact integer arithmetic for precise calculation and floating-point approximation for cross-validation within tolerance bounds. The oracle's

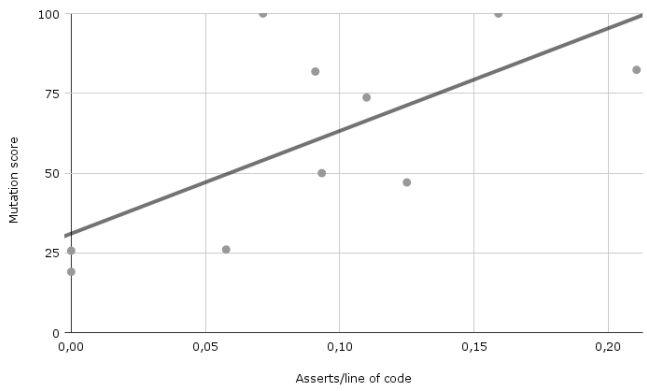

Figure 5: Correlation between mutation score and the number of assertions in the fuzz target

```
FUZZ_TARGET(feefrac_div_fallback)
{
    FuzzedDataProvider provider(buffer.data(), buffer.size());
    auto num_high = provider.ConsumeIntegral<int64_t>();
    auto num_low = provider.ConsumeIntegral<uint32_t>();
    std::pair<int64_t, uint32_t> num{num_high, num_low};
    auto den = provider.ConsumeIntegralInRange<int32_t>(1, std::
        ↪ numeric_limits<int32_t>::max());
    auto round_down = provider.ConsumeBool();

    bool is_negative = num_high < 0;
    auto num_abs = Abs256(num);
    auto den_abs = Abs256(den);
    auto quot_abs = (is_negative == round_down) ?
        (num_abs + den_abs - 1) / den_abs :
        num_abs / den_abs;

    if ((is_negative && quot_abs > MAX_ABS_INT64) || (!is_negative &&
        ↪  quot_abs >= MAX_ABS_INT64)) {
        return;
    }

    auto res = FeeFrac::DivFallback(num, den, round_down);
    assert(res == 0 || (res < 0) == is_negative);
    assert(Abs256(res) == quot_abs);

    long double expect = round_down ? std::floor(num_high *
        ↪ 4294967296.0L + num_low) / den
                                    : std::ceil(num_high * 4294967296.0L
                                        ↪  + num_low) / den;
    if (expect == 0.0L) {
        assert(res >= -1 && res <= 1);
    } else if (expect > 0.0L) {
        assert(res >= expect * 0.9999999999999999L - 1.0L);
        assert(res <= expect * 1.0000000000000001L + 1.0L);
    } else {
        assert(res >= expect * 1.0000000000000001L - 1.0L);
        assert(res <= expect * 0.9999999999999999L + 1.0L);
    }
}
```

Figure 6: Bitcoin Core's feefrac_div_fallback fuzz target

effectiveness derives from the mathematical certainty inherent in division operations, where each input tuple has exactly one objectively correct result, unlike typical software components such as file parsers, network protocols, or user interfaces where correct behavior may be context-dependent or subjective. When mutations change the target function's logic (for example, changing ceiling division to floor division), the discrepancy between the mutated function's output and the independently computed expected result triggers assertion failures, creating a detection mechanism that approaches the theoretical ideal of differential fuzzing applied to mutation testing. This mathematical determinism, combined with comprehensive input space coverage and multiple independent verification layers, explains why this target achieves 100% mutation score despite fuzzing's traditional limitations in generating effective test oracles for arbitrary software systems.

7.3 Metamorphic Relations

Metamorphic testing is a software testing technique that addresses the test oracle problem—situations where it is difficult or impossible to determine the correctness of individual outputs—by identifying metamorphic relations (MRs), which are expected relationships between multiple inputs and their corresponding outputs. Instead of verifying the correctness of a single output, metamorphic testing validates whether the outputs of related inputs follow a known and predictable pattern.

When integrated into fuzz targets for mutation testing, we noticed that metamorphic relations significantly enhance mutant detection capabilities by providing multiple independent verification mechanisms that can identify faults often missed by conventional test oracles. By incorporating metamorphic relations into fuzz targets, it gains the ability to automatically generate diverse test inputs while simultaneously applying multiple correctness criteria, thereby increasing the likelihood of exposing mutant behaviors that violate essential program properties. This approach is particularly effective because fuzzing naturally produces edge cases and boundary conditions that stress-test the metamorphic relations, while the relations themselves provide robust oracles that can detect violations regardless of the specific input values generated.

The synergy between fuzzing and metamorphic testing creates a powerful mutation testing environment where the random input generation of fuzzing complements the systematic property verification of metamorphic relations, resulting in substantially higher mutation scores and more comprehensive fault detection than either technique would achieve independently. Furthermore, this integration addresses the oracle problem in software testing by transforming the challenge from determining correct outputs to verifying mathematical relationships, making it feasible to apply rigorous testing to programs where traditional oracle specification would be impractical or impossible.

8 RQ2 - How does the design of fuzz targets impact their ability to kill mutants?

As mentioned before, removing assertions in the fuzz target resulted in a decrease in mutation scores across targets. One of the fuzz targets dropped from 100% to 0% when removing the assertions. The design of fuzz targets significantly affects their ability to kill mutants, particularly through their influence on input coverage, semantic validation, and the presence of runtime assertions. A fuzz target that merely parses inputs or exercises a limited set of functions may achieve high code coverage without effectively distinguishing between correct and mutated behavior. To increase

mutant detection, the fuzz target must not only trigger the mutated code but also include assertions that act as test oracles, checking that program state and outputs conform to expected semantic properties. These assertions can be derived from pre- and post-conditions, invariants, or domain-specific knowledge about the correct behavior of the program under test. For example, in a cryptographic library, a fuzz target should not only check that decoding succeeds but also assert that encoding followed by decoding returns the original input, or that invalid inputs fail gracefully.

One powerful way to enhance fuzz targets is through the incorporation of metamorphic relations—properties that relate multiple inputs and outputs in a way that remains valid under correct program behavior. These relations allow the fuzz target to detect subtle behavioral changes without requiring an explicit ground truth for every input. For instance, if a sorting algorithm is tested, permuting the input should not affect the final sorted result; if this property fails under mutation, the mutant can be killed. Similarly, in numerical software, scaling input values or changing units might produce predictable transformations in outputs, which can be checked through metamorphic assertions. By embedding such properties directly into the fuzz target, testers can transform implicit correctness expectations into explicit checks that increase the discriminative power of the target. Thus, the inclusion of metamorphic relations, alongside traditional assertions and domain-specific invariants, makes fuzz targets more sensitive to behavioral deviations, substantially improving their ability to detect and kill non-equivalent mutants.

8.1 Longitudinal Studies

Finally, conducting longitudinal studies that track the evolution of fuzz targets and their mutation detection capabilities over time would provide insights into how testing effectiveness changes as software systems evolve. This includes understanding how code changes affect the validity of existing oracles, investigating strategies for maintaining oracle effectiveness as system complexity increases, and developing metrics for monitoring the degradation of fuzz target effectiveness over time. Such studies would inform best practices for maintaining high-quality fuzzing infrastructure throughout the software development lifecycle.

9 Threats to validity

This section discusses the potential threats to the validity of our empirical study and the steps taken to mitigate them. We organize these threats according to the standard classification of internal, external, construct, and conclusion validity.

9.1 Internal Validity

Internal validity concerns the extent to which our experimental design allows us to draw causal conclusions about the relationship between fuzz target design and mutation detection effectiveness.

- Seed Corpus Quality: Our experiments relied on existing seed corpora that have been developed and refined over several years by the Bitcoin Core development team. While these corpora represent high-quality inputs that achieve substantial code coverage, they may not be representative of what a typical fuzzing campaign would generate automatically. This could lead to overestimating the mutation detection capabilities of fuzz targets, as the curated inputs might be particularly effective at triggering specific program behaviors. To partially address this concern, we selected targets with varying corpus sizes and characteristics, though a more comprehensive evaluation would involve generating fresh corpora through automated fuzzing campaigns.
- Mutation Tool Bias: Our choice of mutation-core as the mutation testing tool, while well-suited for Bitcoin Core, introduces potential bias in the types and locations of generated mutants. The tool's design specifically avoids generating "useless" mutants according to Bitcoin Core's nuances, which may result in a mutation set that is either easier or harder to detect than what other mutation testing tools would produce. This could affect the generalizability of our mutation score measurements and the relative performance comparisons between different fuzz targets.
- Target Selection Bias: The selection of 10 fuzz targets from Bitcoin Core was based on our goal of covering diverse functional areas, but this selection process was not entirely systematic. We may have inadvertently chosen targets that are more amenable to mutation detection due to their specific characteristics or implementation patterns. A more rigorous selection process based on random sampling or explicit criteria would strengthen the internal validity of our findings.

9.2 External Validity

External validity addresses the generalizability of our findings to other software systems, domains, and fuzzing approaches.

- Single System Limitation: Our evaluation focused exclusively on Bitcoin Core, a cryptocurrency implementation with specific characteristics that may not be representative of other software domains. Bitcoin Core's emphasis on correctness, extensive use of cryptographic operations, and well-established fuzzing infrastructure may make it more suitable for mutation detection than typical software systems. The applicability of our identified design patterns (round-trip testing, mathematical oracles, metamorphic relations) to domains such as web applications, embedded systems, or user interface software remains uncertain.
- Domain-Specific Findings: The effectiveness of the design patterns we identified may be particularly pronounced in cryptocurrency software due to the prevalence of serialization operations, cryptographic functions, and mathematical computations. These characteristics naturally lend themselves to the oracle types we found most effective. Software in other domains may require different approaches to achieving high mutation scores, limiting the direct transferability of our specific recommendations.
- Programming Language and Ecosystem: Our study was conducted entirely within the C++ ecosystem using Bitcoin Core's specific build system and testing infrastructure. The findings may not apply to other programming languages with different memory management models, type systems, or testing frameworks. Languages with stronger type systems or built-in assertion mechanisms might exhibit different

patterns in the relationship between fuzz target design and mutation detection.

9.3 Construct Validity

Construct validity concerns whether our measurements and experimental setup accurately capture the theoretical concepts we aim to study.

- Mutation Score as Effectiveness Measure: We used mutation score as the primary metric for evaluating fuzz target effectiveness. While mutation score is a well-established measure in software testing research, it may not fully capture all aspects of testing effectiveness. Some mutants might represent unrealistic faults that would never occur in practice, while others might model critical errors that deserve greater weight in evaluation. Additionally, equivalent mutants—those that do not change program behavior—can artificially deflate mutation scores, though mutation-core aims to minimize such cases.
- Assertion Removal Methodology: Our approach of removing all assertions to understand their contribution to mutation detection provides valuable insights but represents a somewhat artificial scenario. In practice, developers would likely replace explicit assertions with alternative verification mechanisms rather than eliminating all forms of behavioral checking. This methodological choice, while useful for isolating the effect of assertions, may not reflect realistic development practices and could overestimate the importance of assertion-based oracles.
- Limited Mutation Operators: Although mutation-core implements multiple mutation operators designed specifically for Bitcoin Core, the set of operators may not comprehensively cover all possible fault types that could occur in real software. The effectiveness of our fuzz targets might vary significantly when evaluated against different mutation operators or fault models, potentially affecting the validity of our conclusions about fuzz target design effectiveness.

9.4 Conclusion Validity

Conclusion validity addresses the statistical and logical soundness of our inferences from the experimental data.

- Limited Statistical Power: With only 10 fuzz targets in our study, our ability to detect statistically significant relationships and draw robust statistical conclusions is limited. The positive correlation we observed between assertion density and mutation score, while suggestive, is based on a relatively small sample size that may not provide sufficient statistical power for definitive conclusions. Larger-scale studies with more targets would be needed to confirm these relationships with greater confidence.
- Confounding Variables: Several variables could potentially confound our results, including the complexity of the functions being tested, the quality of the existing test corpus, and the specific implementation characteristics of each fuzz target. We did not systematically control for these factors, which could influence both mutation scores and the apparent effectiveness of different design patterns. Future studies should consider incorporating these variables into the experimental design or statistical analysis.
- Generalization from Extreme Cases: Some of our strongest conclusions are based on extreme cases, such as the fuzz targets that achieved 100% mutation scores. While these cases provide valuable insights into effective design patterns, they may represent outliers that do not reflect typical performance. The generalization of design principles derived from these exceptional cases to more typical fuzzing scenarios requires additional validation.

Despite these threats to validity, we believe our study provides valuable insights into the relationship between fuzz target design and mutation detection effectiveness. Future research should address these limitations through larger-scale studies, multiple subject systems, and more comprehensive experimental designs to further validate and extend our findings.

10 Conclusion

This paper has presented a comprehensive empirical evaluation of the effectiveness of fuzz testing in detecting mutants, challenging the prevailing assumption that fuzzing performs poorly in mutation testing scenarios. Through our systematic analysis of 10 fuzz targets from Bitcoin Core, encompassing 726 mutants across diverse functional areas, we have demonstrated that well-designed fuzz targets can achieve remarkably high mutation scores, including two cases of 100% mutant detection. Our investigation has revealed three critical design patterns that significantly enhance the mutation detection capabilities of fuzz targets. Round-trip testing approaches, exemplified by serialization-deserialization cycles, provide robust oracles by exploiting the symmetry of paired operations. Mathematical oracles that implement exact behavioral verification through redundant calculations offer deterministic correctness validation for computational functions. Metamorphic relations enable the verification of expected relationships between inputs and outputs, addressing the test oracle problem in domains where explicit correctness criteria are difficult to establish.

The strong positive correlation we observed between assertion density and mutation scores underscores the fundamental importance of explicit oracles in fuzz target design. Our experiments demonstrated that removing assertions led to substantial drops in mutation detection rates, with one target experiencing a complete fall from 100% to 0% effectiveness. This finding emphasizes that effective mutation detection requires not only achieving high code coverage but also incorporating mechanisms that can distinguish between correct and incorrect program behavior.

These results have significant implications for the broader software testing community. First, they suggest that the perceived limitations of fuzzing in detecting subtle behavioral faults may be largely attributed to inadequate test oracle design rather than fundamental shortcomings of the fuzzing approach itself. Second, our findings provide concrete guidance for practitioners seeking to improve their fuzzing campaigns by incorporating domain-specific assertions, metamorphic properties, and round-trip verification mechanisms.

Our work contributes to bridging the gap between fuzzing and mutation testing communities, demonstrating that these complementary approaches can be synergistically combined to achieve more comprehensive fault detection. By revealing the potential of well-designed fuzz targets to detect subtle behavioral deviations, this study opens new avenues for developing more effective automated testing methodologies for safety-critical software systems. The evidence presented here suggests that with thoughtful design and implementation, fuzzing can transcend its traditional role as a crash-finding technique to become a powerful tool for comprehensive behavioral validation in software development practices.

10.1 Future work

Our empirical evaluation of fuzz targets' mutation detection capabilities has revealed promising directions for enhancing the intersection of fuzzing and mutation testing. Several avenues for future research emerge from our findings and the limitations identified in our study.

10.1.1 Automated Oracle Generation. Building on our observation that assertion density strongly correlates with mutation detection effectiveness, future work should investigate automated techniques for generating effective oracles in fuzz targets. This could include developing static analysis tools that identify potential metamorphic relations from program structure, automatically inferring round-trip properties from API usage patterns, or generating mathematical oracles for computational functions. Machine learning approaches could be trained on successful fuzz targets to predict what types of assertions would be most effective for a given function or module. The goal would be to reduce the manual effort required to design effective fuzz targets while maintaining or improving their mutation detection capabilities.

10.1.2 Corpus Evolution. Our study relied on existing, manually curated seed corpora, which may not reflect the typical performance of automated fuzzing campaigns. Future research should investigate how corpus quality and evolution strategies affect mutation detection over time. This includes developing fuzzing strategies that prioritize inputs likely to expose semantic differences rather than just achieving code coverage. Additionally, investigating adaptive corpus curation techniques that retain inputs based on their ability to kill mutants could lead to more effective fuzzing campaigns for comprehensive behavioral validation.

ARTIFACT AVAILABILITY

Artifact is available at https://github.com/brunoerg/bitcoin/tree/SAST.

ACKNOWLEDGMENTS

Simone R. S. Souza is supported by a research grant from the Brazilian funding agency CNPq with reference number: 306719/2025-8.

REFERENCES

[1] Marcio Delamaro, Mario Jino, and Jose Maldonado. 2016. *Introdução ao teste de software - 2ed.* Elsevier Brasil.

[2] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3395363.3397372

[3] Bruno Garcia, Marcio Delamaro, and Simone Souza. 2024. Towards differential fuzzing to reduce manual efforts to identify equivalent mutants: A preliminary study. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 568–573. doi:10.5753/sbes.2024.3557

[4] Rahul Gopinath, Philipp Görz, and Alex Groce. 2022. Mutation Analysis: Answering the Fuzzing Challenge. arXiv:2201.11303 [cs.SE] https://arxiv.org/abs/2201.11303

[5] Alex Groce, Kush Jain, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Looking for Lacunae in Bitcoin Core's Fuzzing Efforts. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 185–186. doi:10.1145/3510457.3513072

[6] Pieter Hartel and Richard Schumi. 2020. Mutation Testing of Smart Contracts at Scale. In *Tests and Proofs*, Wolfgang Ahrendt and Heike Wehrheim (Eds.). Springer International Publishing, Cham, 23–42.

[7] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1158–1161. doi:10.1109/ASE.2019.00126

[8] Gunel Jahangirova. 2017. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 444–447. doi:10.1145/3092703.3098235

[9] 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

[10] Jaekwon Lee, Enrico Vigano, Fabrizio Pastore, and Lionel Briand. 2024. MOTIF: A tool for Mutation Testing with Fuzzing . In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 451–453. doi:10.1109/ICST60714.2024.00052

[11] Amol Saxena, Roheet Bhatnagar, and Devesh Kumar Srivastava. 2021. Improving Effectiveness of Spectrum-based Software Fault Localization using Mutation Testing. In *2021 2nd International Conference for Emerging Technology (INCET)*. 1–7. doi:10.1109/INCET51464.2021.9456109

[12] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[13] Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding Greybox Fuzzing with Mutation Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 929–941. doi:10.1145/3597926.3598107

[14] Pieter Wuille. 2017. BIP 173: Base32 address format for native v0-16 witness outputs. https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki Bitcoin Improvement Proposal.