BitcoinFuzz: Differential Fuzzing of Bitcoin and Lightning Network Implementations

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

Erick Cestari
Vinteum
erickcestari03@gmail.com

Abstract

Bitcoin and the Lightning Network are critical financial infrastructure supported by multiple independent implementations. When these implementations diverge in behavior, the consequences can range from network splits to loss of funds. This paper presents BitcoinFuzz, a differential fuzzing framework designed to uncover behavioral inconsistencies across Bitcoin and Lightning Network implementations. We describe the motivation, methodology, key challenges encountered in practice, and results - including over 60 bugs found across multiple projects. Our experience reveals not only implementation bugs but also gaps in protocol specifications themselves, offering lessons for the broader open-source and software engineering community.

Keywords

Fuzzing, Differential Fuzzing, Bitcoin, Blockchain, Software Testing

1 Introduction

Bitcoin is unusual among large-scale distributed systems in that its protocol is implemented by multiple independent teams in different programming languages. Projects such as [1]Bitcoin Core, [2]btcd, and [3]libbitcoin all aim to implement the same consensus rules, yet subtle divergences can have severe consequences: a node that accepts a block others reject can be forked off the network; one that rejects a valid transaction can silently lose funds. The Lightning Network, a second-layer payment protocol built on top of Bitcoin, faces similar challenges, implementations like LND, CLN, and Eclair must interoperate correctly under all conditions [5]. **With Bitcoin now exceeding $2T in market cap, the industry has emerged with the concern of making these implementations more secure and reliable.**

Differential fuzzing is a natural fit for this problem. Instead of checking a single program against a known-good oracle, differential fuzzing feeds the same inputs to multiple implementations and flags any divergence in output or behavior. This technique has proven highly effective in compilers, databases, and network stacks, but its application to Bitcoin and Lightning has been largely unexplored. Groce et al. [3] pointed out that since the Bitcoin protocol does not have a specification, differential fuzzing might be promising, given that a reference implementation is such a specification.

This paper describes **BitcoinFuzz**, an open-source differential fuzzing framework targeting Bitcoin and Lightning implementations. We present the design rationale, the practical challenges we encountered during development and deployment, and a summary of the bugs we discovered. Our goal is both to share a useful tool with the community and to document the broader lessons learned about protocol specification quality and open-source security practices.

[1]https://github.com/bitcoin/bitcoin
[2]https://github.com/btcsuite/btcd
[3]https://github.com/libbitcoin/libbitcoin-node

2 Background

2.1 Differential Fuzzing

Differential fuzzing extends traditional fuzzing by using behavioral equivalence between multiple programs as the test oracle. Rather than needing a formal specification of correct behavior, the fuzzer identifies discrepancies between implementations and surfaces them for manual inspection. This approach has been used effectively in projects like OSS-Fuzz against compiler toolchains, libraries, compilers [1] [6].

Another successful case of differential fuzzing is the [4]cryptofuzz project. It applies differential fuzzing to cryptography libraries, employing coverage-guided libFuzzer as its core, and has been finding many bugs and vulnerabilities. Based on *cryptofuzz*, Jin et al. [4] propose to improve the performance of differential fuzzing combining fuzzing and concolic execution.

2.2 Multiple Implementations and Consensus Risk

Bitcoin has historically multiple full node implementations, even though Bitcoin Core dominates the network. This diversity is seen as a strength for decentralization, but it introduces a critical requirement: all implementations must agree on consensus rules with absolute precision. A bug that causes even one implementation to accept or reject a block differently can split the network, as occurred in the 2013 Bitcoin fork caused by a database-level divergence between versions.

2.3 The Lightning Network and BOLT Specifications

The Lightning Network is governed by the [5]BOLT (Basis of Lightning Technology) specification, a set of documents collaboratively maintained by implementation teams. Unlike Bitcoin's more informal specification process, BOLT provides relatively detailed protocol descriptions. However, as we will discuss, formal documentation does not guarantee complete coverage of edge cases [5].

3 BitcoinFuzz

BitcoinFuzz is a differential fuzzing harness that wraps multiple Bitcoin and Lightning implementations behind a common interface.

[4]https://github.com/MozillaSecurity/cryptofuzz
[5]https://github.com/lightning/bolts

Fuzz inputs are generated and mutated using a coverage-guided engine, then fed simultaneously to all targets. Any divergence triggers a finding for human review. The user can choose which modules (implementations that will be targeted) they want to compile and fuzz.

The framework currently targets most of Bitcoin and Lightning implementation, including Bitcoin Core, rust-bitcoin, btcd, LND, Eclair, and covers input surfaces including script deserialization, transaction validation, block parsing, Lightning message handling and more.

4 Challenges

Deploying differential fuzzing against real-world, production Bitcoin and Lightning implementations surfaced several non-trivial challenges that go beyond typical fuzz testing.

4.1 Unresolved Bugs Block Fuzzing Progress

When BitcoinFuzz discovers a divergence and reports it to the upstream project, forward progress depends on the project acknowledging and fixing the issue. In practice, when a reported bug is not fixed, whether due to disagreement about whether it is a bug, low prioritization, or slow release cycles, it creates a persistent false positive in the fuzzer output. Every subsequent run re-triggers the same finding, making it harder to identify new bugs buried in the noise. Managing this backlog and deciding when to suppress known findings without masking real regressions became a significant operational burden.

4.2 Bitcoin's Informal Specification

Bitcoin does not have a single authoritative protocol specification. The de facto specification is Bitcoin Core's behavior, supplemented by Bitcoin Improvement Proposals (BIPs) [2], which vary widely in precision and completeness. When BitcoinFuzz finds a divergence, it is often not immediately clear which implementation is correct - or whether the question even has a definitive answer. Several of our findings required extended discussions with developers from multiple implementations before a consensus on expected behavior could be reached. This reflects a systemic gap: without a rigorous spec, even well-intentioned implementations can diverge in ways that are difficult to adjudicate.

4.3 BIP Updates and Implementation Lag

The BIP process allows proposals to be revised after initial adoption. We found cases where a BIP was updated to clarify or change required behavior, but only one implementation incorporated the change while others continued following the older version. From a fuzzing perspective, this produces a real divergence that is technically a bug in the lagging implementations, even if it has not yet caused observed problems on the network. These cases highlight the need for better coordination mechanisms across implementation teams when specifications are revised.

4.4 Lightning Spec Gaps Despite BOLT Coverage

The Lightning Network's BOLT specifications are considerably more formal than Bitcoin's protocol documentation, and we expected them to reduce ambiguity. However, several bugs we found in Lightning implementations pointed to situations not covered, or only ambiguously covered, by BOLT. Edge cases in message sequencing, error handling, and channel state transitions were areas where implementations had made different reasonable-sounding interpretations of under-specified behavior. This suggests that even a well-maintained spec can leave meaningful gaps when the protocol's state space is large.

4.5 Maintaining Forks of the Projects

Effective differential fuzzing of Bitcoin and Lightning implementations requires more than simply feeding identical inputs to each target. To maximize code coverage and avoid wasting fuzzing cycles on computationally expensive operations that are orthogonal to the logic under test, it is often necessary to mock or stub out certain components. The most prominent example is cryptographic signature verification. In normal operation, every Bitcoin transaction and Lightning message undergoes one or more ECDSA or Schnorr signature checks. These checks are correct by construction for any input the fuzzer generates, since the fuzzer cannot produce valid signatures for arbitrary data. Leaving them enabled means each fuzzing iteration spends significant CPU time on verification that will almost always succeed trivially or fail trivially, without exercising any interesting protocol logic. Disabling or stubbing signature verification allows the fuzzer to explore transaction validation, script evaluation, and message handling code paths at far greater depth and speed. Similar arguments apply to other expensive or determinism-breaking operations: proof-of-work validation, timestamp checks, network I/O, and calls to system randomness sources. A fuzzing harness that bypasses these produces faster iterations and more reproducible behavior, both of which are essential for a practical differential fuzzing campaign.

The challenge is that the ease of introducing such mocks varies enormously across implementations. Bitcoin Core, written in C++, exposes compile-time flags and a relatively modular architecture that makes it possible to disable signature verification and other checks without invasive changes. Other implementations, however, were not designed with fuzzing in mind. Their verification logic may be tightly coupled to surrounding code, may lack any abstraction boundary that can be stubbed, or may be written in a language or framework where dependency injection is cumbersome.

When clean mocking is not achievable through configuration or a thin shim layer, the only practical option is to maintain a fork of the upstream project with targeted modifications. These forks are deliberately minimal: the goal is to change as little as possible beyond the specific instrumentation needed for fuzzing, so that the forked implementation remains a faithful proxy for the real one in all respects that matter to the differential test.

5 Results

To date, BitcoinFuzz has identified over 60 bugs across Bitcoin and Lightning implementations. These range from divergences in script validation, transaction parsing edge cases, and p2p message handling, to Lightning-specific issues in wire-message parsing, invoice and offer decoding, and onion-packet handling. A portion of these bugs have been confirmed and fixed by the respective upstream projects; others remain open or under discussion. Several findings

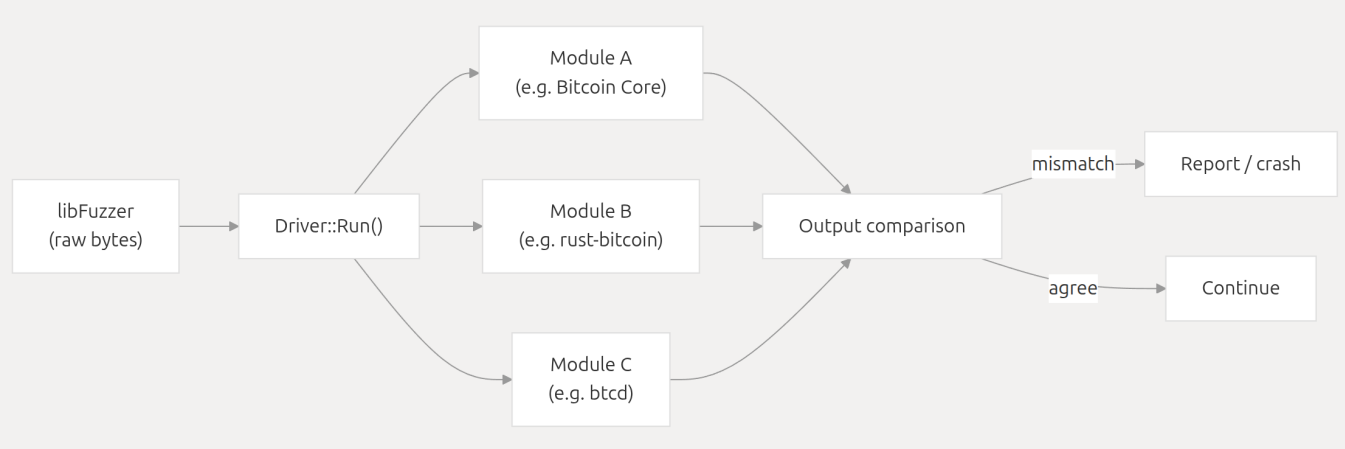

Figure 1: Bitcoinfuzz architeture, via DeepWiki. (https://deepwiki.com/bitcoinfuzz/bitcoinfuzz).

led directly to improvements in the BOLT specifications rather than, or in addition to, fixes in implementation code, demonstrating that differential fuzzing can serve as a specification validation tool as much as an implementation testing tool.

6 Lessons Learned and Discussion

Our experience with BitcoinFuzz surfaces lessons relevant to the broader software engineering community, particularly for safety-critical open-source infrastructure.

First, differential fuzzing is most effective when combined with a robust bug triage and coordination process across projects. The technical work of finding bugs is only part of the problem; the organizational challenge of getting them fixed is equally important.

Second, informal specifications create real security risks. The Bitcoin ecosystem's reliance on Bitcoin Core as the reference implementation is pragmatic but fragile. The community would benefit from more investment in formal or semi-formal protocol documentation.

Third, specifications and implementations should be co-evolved. When a specification is updated, there should be a clear mechanism to notify all implementation teams and verify compliance. BitcoinFuzz can play a role here by running targeted differential tests after any BIP revision.

7 Conclusion

BitcoinFuzz demonstrates that differential fuzzing is a practical and valuable technique for finding bugs in Bitcoin and Lightning Network implementations. With over 60 bugs found, the project has already had tangible impact on the security of the ecosystem. More broadly, our experience highlights that implementation correctness in multi-implementation protocols depends not only on engineering discipline but also on the quality and maintainability of the underlying specifications. We release BitcoinFuzz as open source and invite contributions from the community.

Artifact Availability

The BitcoinFuzz tool is publicly available at https://github.com/bitcoinfuzz/bitcoinfuzz.

References

[1] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 351–364. doi:10.1145/3548606.3560624

[2] Bitcoin Developers. 2026. Bitcoin Improvement Proposals (BIPs). https://github.com/bitcoin/bips. Accessed: 2026-06-05.

[3] 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

[4] Hoyong Jin, Dohyeon An, and Taekyoung Kwon. 2023. Differential Testing of Cryptographic Libraries with Hybrid Fuzzing. In *Information Security and Cryptology – ICISC 2022*, Seung-Hyun Seo and Hwajeong Seo (Eds.). Springer Nature Switzerland, Cham, 124–144.

[5] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

[6] Alessandro Sorniotti, Michael Weissbacher, and Anil Kurmus. 2023. Go or no go: Differential fuzzing of native and C libraries. In *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE, 349–363.