Bitcoin Core

Technical Security Audit Report

Reference 25-05-2133-REP
Version 1.3
Date 2025-11-14

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

Quarkslab

1. Project information

1.1. Document history

Version	Date	Details	Authors
1.0	2025-10-14	Initial audit	Robin David, Nicolas Surbayrole, Mihail Kirov
1.2	2025-11-10	Updated version	Robin David, Nicolas Surbayrole, Mihail Kirov
1.3	2025-11-14	Updated version	Robin David, Nicolas Surbayrole, Mihail Kirov

1.2. Contacts

1.2.1. Quarkslab

Name	Role	Email
Frédéric Raynal	CEO	fraynal@quarkslab.com
Robin David	Security R&D Engineer	rdavid@quarkslab.com
Nicolas Surbayrole	Security R&D Engineer	nsurbayrole@quarkslab.com
Mihail Kirov	Security R&D Engineer	mkirov@quarkslab.com
Pauline Sauder	Project Manager	psauder@quarkslab.com

1.2.2. OSTIF

Name	Role	Email
Derek Zimmer	Executive Director (OSTIF)	derek@ostif.org
Amir Montazery	Managing Director (OSTIF)	amir@ostif.org
Helen Woeste	Communications Manager (OSTIF)	helen@ostif.org

1.2.3. Brink

Name	Role	Email
Mike Schmidt	Executive Director Brink	
Niklas Goegge	Security Engineer	

1.2.4. Chaincode Labs

Name	Role	Email
Antoine Poinsot	Software Engineer	

About the OSTIF

The **Open Source Technology Improvement Fund (OSTIF)** is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, government, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landscape.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of dollars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their security posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience positions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is ostif.org. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at `contactus@ostif.org` or our Github.

- Derek Zimmer, Executive Director
- Amir Montazery, Managing Director
- Helen Woeste, Communications and Community Manager
- Tom Welter, Project Manager

Contents

2. Executive Summary

Note: Metric definition and vulnerability classification are detailed in the reading guide (Chapter 3).

2.1. Context

The Open Source Technology Improvement Fund, Inc (**OSTIF**), on the behalf of Brink, has commissioned Quarkslab to perform a security audit of Bitcoin Core [1], the reference implementation of the Bitcoin protocol [2].

The OSTIF and Quarkslab have collaborated on several security assessments over the years, in the context of securing widely used and crucial open-source projects, such as:

- PHP security audit, 2025
- Operator Fabric security audit, 2024
- Cloud Native Buildpacks security audit, 2024
- Kuksa security audit, 2024

The assessment is motivated by the criticity of the Bitcoin Core code, securing an asset valued at $2,303 billion USD[1], but more specifically by the lack of any external third-party audit of the Bitcoin Core codebase.

This report is the **first public security audit of Bitcoin Core**, and it is the result of a collaborative effort between Quarkslab, the OSTIF, Brink, Chaincode Labs and the Bitcoin Core developers.

The assessment was carried over a 104 days period from May 2025 to September 2025. A specific commit version was chosen as base of the audit af65fd1a333011137dafd3df9a51704fd319feb4.

This report presents the results of the security assessment.

2.2. Objectives

The security assessment aimed at assisting Bitcoin Core's core developers and the community in assessing and strengthening the project's security practices especially on the testing and fuzzing part. Beyond identifying security issues, the goal is to provide Bitcoin Core developers key actionable insights and utilities to further improve the codebase's security.

The codebase was analyzed within a defined scope, which was established and agreed upon by both Brink's developers and Quarkslab. The exact scope of work is detailed in Section 4.5. Based on this scope and the allocated time frame for the audit, an attack model was defined and the following tasks were defined:

[1] as of July 2025

- identifying security issues in the P2P *(most critical components)* and the impacted components e.g: mempool, coincache etc.;
- basic tooling evaluation;
- fuzzing harnesses evaluation and improvement;
- manual code review of the codebase, with a focus on the attack model and specific especially thread management;
- improving Static Application Security Testing (SAST) tooling to enhance existing Github CI automated workflows;
- building new fuzzing harnesses compatible with the current infrastructure using OSS-fuzz for potential code segments that are not currently covered;

2.3. Methodology

Bitcoin Core is a very large and complex codebase, with over 200K LoC (lines of code) and hundreds of contributors since the first version published by Satoshi Nakamoto [3]. To assess the security Quarkslab's team first needed to familiarize themselves with the project structure, internal components and associated security concerns.

The evaluation employed a combination of static analysis and dynamic testing. The dynamic approach relied both on existing unit, integration and fuzz tests already in production, but also aimed to complement it with new harnesses or approaches. Dynamic testing complements the static review and enables validating or refuting hypotheses gathered during static analysis.

The following methodology and roadmap was presented to Brink and Chaincode's Bitcoin Core developers and started after approval:

1. **Step 1:** Discovery & Threat modeling
   - The first step is to gather and review the available documentation and project resources around Bitcoin Core implementation. These corpora are scattered in different projects and websites. Thanks to Brink's team, Quarkslab security auditors were onboarded on the codebase and various security aspects during a couple of days in their office. It allowed them to refine the threat model and to identify the components to include in the scope with a better understanding of the features to be evaluated.
2. **Step 2:** Manual code review & Static analysis
   - The manual code review focused on scrutinizing the source code to identify vulnerabilities related to the implementation and logic of specific components harder to assess dynamically like thread management. Some static analyzers could be used to assists in identifying potential issues.
3. **Step 3**: Improving current testing
   - This step involves reviewing the current testing framework and identifying potential improvements. This includes evaluating the existing unit tests, integration tests, fuzzing harnesses and the resulting coverage of these approaches. The goal is to enhance the current testing framework to cover more code paths and edge cases while remaining compatible with the current infrastructure based on libFuzzer and `oss-fuzz`.

4. **Step 4**: Exploring new testing approaches
   - This step involves researching and proposing new testing methodologies or tools in order to identify gaps in the current testing strategy. Integration within current codebase, while desirable, is not the key focus. This may include differential fuzzing, ensemble fuzzing, property-based testing, or other innovative approaches.
5. **Step 5**: Report Writing
   - The current security state and all findings, recommendations, and action plans are compiled into a comprehensive report for OSTIF and any other stakeholders. The report details all the covered aspects agreed upon in the scope of work.

Legal Notice

This report reflects the work and results obtained within the duration of the audit on the specified scope and as agreed between Brink and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure the code is bug or vulnerability free. Also, comments and observations reflect the state of the code spanning from May to September 2025.

2.4. Deliverables & Findings Summary

As agreed in the scope of work, a thorough assessment of the P2P part was performed with a strong focus on assessing and improving the testing infrastructure in place. During the time frame of the security audit, Quarkslab has discovered 2 low security issues and 13 suggestional improvements. None of them have a security impact.

Besides finding weaknesses, the audit also aimed at improving the overall security posture of Bitcoin Core. To this end, Quarkslab has experimented and produced a number of harnesses, data and utilities to improve the testing. Notably, Quarkslab was able to trigger **chain reorganization** events through fuzzing. This event had never been triggered by existing harnesses.[2] Additional deliverables are the following:

- Better fuzzing coverage of existing harnesses with additional lines covered
- New test inputs covering new code paths and events
- 3 new harnesses testing block connection and chain reorganisation, that will be submitted as merge requests to the main codebase
- a virtual Filesystem utility class to be used in fuzzing harnesses for fast state restoration
- 14 structured fuzzing harnesses using libprotobuf-mutator *(+1 grammar-based for miniscript)*
- Non regression utility script for differential testing based on tracepoints and a new macro called `SERIALIZE_TO_DATATRACE`
- 2 differential testing harnesses for `chacha20_poly1305` and `SHA256` variants (SSE4, SHANI)
- a Docker image to run harnesses in an ensemble fuzzing setting with PASTIS *(using AFL++, Honggfuzz and libFuzzer)*

ID	Name	Perimeter
LOW-2	Use of `PT_GUARDED_BY` instead of `GUARDED_BY` in `CBlockPolicyEstimator`	ThreadSafety
LOW-3	Risky usage of `Assume` to check null division	Fee Calculation
INFO-1	Add `GUARDED_BY` attribute to additionnal members	ThreadSafety
INFO-4	No pre-determined or hardcoded key materials to spend coins	Fuzzing Harnesses
INFO-5	`bip324_cipher_roundtrip`: fuzz highly entropic bytes	Fuzzing Harness
INFO-6	Lack of exercising results for multiple harnesses	Fuzzing Harness
INFO-7	`ellswift_roundtrip`: fuzz highly entropic bytes	Fuzzing Harness
INFO-8	`descriptor_parse`: implementation impedes fuzzing coverage readability	Fuzzing Harness
INFO-9	`parse_iso8601`: Cast `int32_t` into `int64_t`	Fuzzing Harness
INFO-10	`primitive_transaction`: Assertions are tautologies	Fuzzing Harness
INFO-11	Using FuzzingDataProvider in `script_flags`	Fuzzing Harness

[2] Reorganization are covered by integration/functional tests and might be covered by external third-party tooling.

ID	Name	Perimeter
INFO-12	`socks5`: field `username`, `password` needlessly too long	Fuzzing Harness
INFO-13	Reduce `LIMITED_WHILE` iterations	Fuzzing Harness
INFO-14	Limit input size for fuzzing	Fuzzing Harness
INFO-15	Reading highly entropic fields from the input	Fuzzing Harness

2.5. Recommendation and Action Plan

The suggested actions to take for the various findings are summarized in the table below.

ID	Recommendation
LOW-2	The `feeStats`, `shortStats` and `longStats` members of `CBlockPolicyEstimator` should be protected with `GUARDED_BY`. The lock of `m_cs_fee_estimator` in `CBlockPolicyEstimator::estimateRawFee()` must be moved to the beginning of the method.
LOW-3	It is recommended to replace `Assume()` with an `Assert()` to ensure that this critical check is performed in both `DEBUG` and `RELEASE` modes.
INFO-1	Add `GUARDED_BY` for non-atomic, non-constant class members that are already implicitly covered by a mutex.
INFO-4	It is recommend to help fuzzing by casually providing it materials like keys, coins known *a priori*, that can be used by the harness to generate valid transactions or blocks. Another option is to generate coins that do not requires any keys to be spent.
INFO-5	Avoid reading high-entropy bytes directly from the fuzz input. Because private keys and seeds are effectively random, embedding them wastes mutational effort. They should instead be be drawn from a secure random source.
INFO-6	Ensure results of computations and function calls are validated via assertions or additional checks.
INFO-7	Avoid reading high-entropy bytes directly from the fuzz input. Because private keys and seeds are effectively random, embedding them wastes mutational effort. They should instead be drawn from a secure random source.
INFO-8	While the harness is written in a very compact form, it is recommended that, for post-fuzzing analysis, the code be made more self-explanatory. In this case, unrolling the for loop into two distinct loops—one with `require_checksum` set to true and the other to false—would improve readability and make the results easier to interpret.
INFO-9	The harness should directly consume a 64-bit integer.
INFO-10	The contents of the parsed transaction should be actively manipulated and tested using meaningful checks rather than relying on tautological assertions.
INFO-11	Modifying the harness to use `FuzzingDataProvider` in order to be more consistent and uniform with the other harnesses.
INFO-12	Reduce both field sizes to facilitate fuzzer mutations and improve exploration.
INFO-13	Reducing the number of iterations to a more reasonable value, for example to 1000.
INFO-14	It is recommended to limit input sizes to a reasonable value (e.g., 8 kB). A harness that requires very large inputs to trigger new behaviors is likely suboptimal, as libFuzzer persistent harnesses are designed to iterate inputs very quickly. This

ID	Recommendation
	limitation can be further mitigated by regularly merging and trimming existing test cases to maintain efficiency and avoid unnecessary overhead.
INFO-15	It is recommended to avoid reading highly entropic fields from the input directly or to strongly limit its size.

2.6. Conclusion

Quarkslab has been mandated on behalf of the OSTIF to perform the first public security audit of Bitcoin Core performed by an audit firm. The value at stake secured by the Bitcoin Core protocol requires the strictest and highest security standards possible.

While understanding core concepts of Bitcoin is fairly easy, digging in the implementation of all the BIPs (Bitcoin Improvement Proposals) is a daunting task. Brink and Chaincode Labs engineers have been supportive in providing context and guidance to the Quarkslab auditors who faced a large codebase, yet the most mature and well tested they ever assessed. They also witnessed the contribution process which, while being strict and conservative, remains welcoming to external contributions and ensures best practices are followed before merging code.

The security assessment focused on a specific scope, the P2P part and on most impactful attack scenarios altering consensus or protocol availability. No high-impact issues were found, but marginal gain was brought on existing fuzzing harnesses as well as new ones to cover untested scenarios like chain reorganization. Also, some alternative testing approaches were explored like ensemble fuzzing and differential testing. While not exhibiting any issues in the current code, such approaches can certainly add value to the whole testing strategy and project robustness. In that regard, the snapshot fuzzing approach currently developed by Brink is the most valuable path to explore in order to trigger deeper and more complex bugs.

3. Reading Guide

This reading guide describes the different sections present in this report and gives some insights about the information contained in each of them and how to interpret it.

3.1. Executive summary

The executive summary Section 2 presents the results of the assessment in a non-technical way, summarizing all the findings and explaining the associated risks. For each vulnerability, a severity level is provided as well as a name or short description, and one or more mitigation, as shown below.

ID	Name	Category
CRITICAL	Vulnerability Name #1	Injection
HIGH	Vulnerability Name #2	Remote code injection
MEDIUM	Vulnerability Name #3	Denial of Service
LOW	Vulnerability Name #4	Information leak

Each vulnerability is identified throughout this document by a unique identifier `<LEVEL>-<ID>`, where `ID` is a number and `LEVEL` the severity (`INFO`, `LOW`, `MEDIUM`, `HIGH` or `CRITICAL`). Every vulnerability identifier present in the vulnerabilities summary table is a clickable link that leads to the corresponding technical analysis that details how it was found (and exploited if it was the case). Severity levels are explained in Section 3.2.

The executive summary also provides an action plan with a focus on the identified *quick wins*, some specific mitigation that would drastically improve the security of the assessed system.

3.2. Metric definition

This report uses specific metrics to rate the severity, impact and likelihood of each identified vulnerability.

3.2.1. Impact

The impact is assessed regarding the information an attacker can access by exploiting a vulnerability but also the operational impact such an attack can have. The following table summarizes the different levels of impact we are using in this report and their meanings in terms of information access and availability.

CRITICAL	Allows a total compromise of the assessed system, allowing an attacker to read or modify the data stored in the system as well as altering its behavior.

HIGH	Allows an attacker to impact significantly one or more components, giving access to sensitive data or offering the attacker a possibility to pivot and attack other connected assets.
MEDIUM	Allows an attacker to access some information, or to alter the behavior of the assessed system with restricted permissions.
LOW	Allows an attacker to access non-sensitive information, or to alter the behavior of the assessed system and impact a limited number of users.

3.2.2. Likelihood

The vulnerability likelihood is evaluated by taking the following criteria in consideration:

- **Access conditions**: the vulnerability may require the attacker to have physical access to the targeted asset or to be present in the same network for instance, or can be directly exploited from the Internet.
- **Required skills**: an attacker may need specific skills to exploit the vulnerability.
- **Known available exploit**: when a vulnerability has been published and an exploit is available, the probability a non-skilled attacker would find it and use it is pretty high.

The following table summarizes the different level of vulnerability likelihood:

CRITICAL	The vulnerability is easy to exploit even from an unskilled attacker and has no specific access conditions.
HIGH	The vulnerability is easy to exploit but requires some specific conditions to be met (specific skills or access).
MEDIUM	The vulnerability is not trivial to discover and exploit, requires very specific knowledge or specific access (internal network, physical access to an asset).
LOW	The vulnerability is very difficult to discover and exploit, requires highly specific knowledge or authorized access

3.2.3. Severity

The severity of a vulnerability is defined by its impact and its likelihood, following the following table:

		Impact			
		●●●●	●●●○	●●○○	●○○○
Likelihood	●●●●	CRITICAL	CRITICAL	HIGH	MEDIUM
	●●●○	CRITICAL	HIGH	HIGH	MEDIUM
	●●○○	HIGH	HIGH	MEDIUM	LOW
	●○○○	MEDIUM	MEDIUM	LOW	LOW

4. Introduction

4.1. Context

Bitcoin Core is the reference implementation of the Bitcoin protocol used by most nodes on the network. The code, developed primarily in C++, is maintained by a large community of contributors and follows a strict peer-review process. It undergoes continuous evolution to introduce new features, improve modularity, and enhance performance and security against various classes of attacks (cf. Appendix B.).

The project encompasses all the components required to operate and interact with the Bitcoin network. These include the node software that enforces consensus rules, mining functionality for block production, and a wallet that enables users to manage funds and perform transactions.

While the global state of the blockchain is secured with a proof-of-work [4] consensus mechanism, the Bitcoin peer-to-peer protocol operates in a trustless environment. Consequently, all nodes are considered untrusted, and any messages exchanged between them must be treated as potentially malicious. The codebase is therefore written in a defensive manner, incorporating various mechanisms and design patterns to mitigate denial-of-service (DoS) attacks, network partitioning, and other potential threats.

However, the Bitcoin Core codebase has never undergone a comprehensive third-party security audit. To address this, the OSTIF [5] and Brink [6] have collaborated to fund and coordinate an independent assessment, managed by the OSTIF and conducted by Quarkslab, a company specializing in software security evaluations.

4.2. Goals

The implementation has demonstrated robustness and maturity over the years. The objective of the audit is therefore to provide a security-oriented external perspective, with a strong focus on the most critical components identified in the attack model. The tasks are twofold:

- **Identifying security issues and vulnerability** through static and dynamic analysis of selected components (e.g., mempool (memory pool) [7], reorg (chain reorganization) [8] handling, etc.)
- **Supporting developers in strengthening their security posture** by assessing and implementing new test harnesses and developing new tools to enhance the overall security of the codebase.

In addition to the final security report, some of the proposed improvements may also be submitted directly as merge requests to the main codebase.

4.3. Prerequisites

Nearly all required materials are publicly available, including the open-source codebase released under the MIT license. As agreed with the Bitcoin Core engineers, Bitcoin Core on master branch at the following commit ID is used:

```
af65fd1a333011137dafd3df9a51704fd319feb4
```

This commit has been pushed on May 20th, 2025 and inherits from v29.0 which was released on April 11th, 2025. As part of the audit, a dedicated communication channel was established to facilitate discussions between the Quarkslab team, Brink, and Chaincode Labs.

4.4. Threat Modeling

From a threat modeling perspective, the Bitcoin Core codebase is a large and complex system and exposes two primary attack surfaces:

RPC (JSON-RPC) - The main interface for interacting with a Bitcoin node and performing various operations, such as submitting transactions from wallets, querying the mempool, broadcasting mined blocks, etc. The JSON-RPC is listening on port 8332. The main functionalities include:

- **Blockchain**: Allows querying the blockchain and mempool state, scanning blocks, and retrieving block data.
- **Transactions**: Supports decoding, signing, and broadcasting transactions.
- **Utilities**: Provides functions such as estimating current fee rates, deriving addresses, and signing or verifying messages.
- **Mining**: Facilitates obtaining block templates for mining and submitting mined blocks.
- **Network**: Enables retrieving information about connected nodes, managing banned IPs, and adding new peers.
- **Wallet**: Supports creating, managing, and using wallets, as well as sending transactions.

A significant portion of the codebase is accessible through the RPC interface, representing a notable attack surface for a node. As agreed with the Bitcoin Core developers, Bitcoin's security model places the responsibility for securing the RPC interface on the node operators. Exposing the RPC interface should be done with caution and is not the primary focus of this audit. In practice, very few nodes on the Bitcoin network expose their RPC interfaces. Consequently, **RPC interfaces are considered out of scope for this audit**.

P2P - The peer-to-peer interface is the primary communication channel between nodes and is essential for executing the core blockchain logic and maintaining consensus. For mainnet, the listening port is 8333. The protocol between nodes is bitcoin specific protocol. Pending transactions and blocks are propagated through this interface, and new node discovery also occurs over these connections. During normal operation, a node maintains approximately 125 concurrent connections with peers with a maximum of 8 outbound and 117 inbound connections.

As previously mentioned, peers are considered untrusted, and all messages exchanged must be treated accordingly. These messages directly affect the chain state, mempool, and other internal components. As agreed with the Brink team, **the P2P interface and its underlying components constitute the main focus of this audit**.

4.5. Scope of Work

The security assessment primarily focuses on the P2P interface and the components it affects. These components include:

- Mempool management
- Block validation, and consequently, transaction evaluation
- Chain state management, including scenarios with multiple active tips
- Peer management, such as address handling and ban management

More generally, all 34 different message types (cf. Section 5.2) that can be exchanged between any two peers will be reviewed. The review will be conducted both manually and dynamically through automated testing (fuzzing).

The audit aims to identify memory corruption, DoS, fairness, or consensus issues. Past security advisories [9] illustrate several cases of potential DoS attacks, which are particularly critical in decentralized networks. Such vulnerabilities could lead to network partitioning or a complete takedown of the Bitcoin network given that Bitcoin Core is by far the predominant implementation.

Additionally, attention will be given to areas identified by Bitcoin Core developers as weak or undertested. This includes the clustered mempool implementation [MR28676], which is not part of the frozen commit used for the audit.

4.6. Bitcoin Core Community Development

Bitcoin development is open to all contributors. Numerous resources are available to help newcomers get started with Bitcoin Core[3,4,5]. Additional resources provide explanations of the core concepts and components of the protocol[6,7].

Contributions are submitted as Pull Requests (PRs) and require thorough review and approval from multiple community members before being merged[8]. Discussions about BIPs, MRs, and protocol upgrades take place on the Delving Bitcoin forum ([10]).

Among the ongoing significant changes are the new clustered mempool architecture (PR28676) and `libbitcoinkernel` [11], which aims to modularize the codebase into one or more

[3] https://medium.com/@amitiu/onboarding-to-bitcoin-core-7c1a83b20365
[4] https://www.lopp.net/bitcoin-information.html
[5] https://jameso.be/dev++2018/
[6] https://github.com/chaincodelabs/bitcoin-curriculum
[7] https://chaincode.gitbook.io/seminars/bitcoin-protocol-development/welcome-to-the-bitcoin-protocol
[8] https://bitcoincore.reviews/

core libraries that utility programs can build upon. This refactoring would, for example, allow writing bindings to Bitcoin's consensus and policy rules in other programming languages. This is a long-term project and is excluded from the current audit scope, which focuses on analyzing the current as-is implementation.

4.7. Current State of Testing

The Bitcoin project demonstrates impressive maturity in its testing infrastructure, with over **1200** tests spanning unit tests, integration tests, and fuzzing harnesses. The table below summarizes the distribution of these tests:

Unit-test	Integration Test	Fuzzing Tests
657	322	221

Table 1: Number of tests per category

4.7.1. Unit and functional testing

Bitcoin uses the Boost test framework [12] for unit testing. Tests are located in the `src/test/` directory and registered using the CMake `add_test` command. They can either be run using `ctest --test-dir build/` or `make test`[9]. The test binary `test_bitcoin` can also be run directly to test the different components independently. The current test suite grouped by components is given in Appendix A..

Functional, integration tests are located in `test/function` and work by interacting with `bitcoind` and `bitcoin-qt` node through RPC and P2P. They can be run individually or all together using `build/test/functional/test_runner.py`. The tested components are listed in the table below, comprising a total of 318 tests.

Component	#tests
feature	55
mempool	20
wallet	110
p2p	64
mining	4
rpc	45
interface	11
tool	8
example	1

Table 2: Number of functional tests

There are also extended tests, which can be enabled by passing the `--extended` argument to the test runner script. These tests are not executed by default, as they take longer to

[9] https://bitcoincore.reviews/

run, and they add 4 additional tests, bringing the total to 322. Detailed documentation for functional testing is available in the test/functional/README.md file. These tests cover all major components, resulting in very satisfactory overall test coverage.

4.7.2. Fuzzing

Fuzzing is performed using libFuzzer ([13]), part of the LLVM project, with the `LLVMFuzzerTestOneInput` function. The fuzzing harnesses are located in `src/test/fuzz` and are grouped into a single fuzzing target, `fuzz.c`. The specific harness to use is specified through the `FUZZ_TARGET` environment variable, selecting a single fuzzing binary. Each fuzzing target is compiled with both AddressSanitizer (ASan) [14] and UndefinedBehaviorSanitizer (UBSan) [15] to detect memory and undefined behavior issues during testing.

Warning

Having a single fuzzing binary is convenient for coverage generation but incurs some limitations. For instance, automatic dictionary generation embedded in AFL++ will generate a single dictionary file, while it would make more sense to have a single dictionary per fuzzing target.

The test utility `test_runner.py` runs all fuzzing targets sequentially. The codebase currently contains **221 fuzzing harnesses** that exercise nearly every component (see Chapter 6 for details). Full documentation is available in the fuzzing.md file.

Fuzzing compute resources are provided by OSS-Fuzz [16] and therefore run on Google's infrastructure, which requires using libFuzzer [13]. Alternatives such as AFL++ are only loosely supported and maintained. OSS-Fuzz also supplies automatic vulnerability tracking, statistics, and coverage reports via its dashboards and storage[10,11,12].

Beyond this standard setup, Niklas Goegge, member of Brink, has explored several complementary approaches to improve fuzzing effectiveness. These include a descriptor format for the serialization used in Bitcoin [17], a differential-fuzzing approach [18], structured fuzzing for existing harnesses [19], and a new snapshot-based fuzzer with a grammar encoding of P2P scenarios [20]. Given the complexity of P2P exchanges, snapshot- and grammar-based techniques are a particularly promising way to drive deeper code paths and realistic scenarios. These initiatives have already uncovered robustness issues — and in some cases security vulnerabilities.

[10] https://issues.oss-fuzz.com/issues?q=bitcoin-core
[11] https://introspector.oss-fuzz.com/project-profile?project=bitcoin-core
[12] https://storage.googleapis.com/oss-fuzz-coverage/bitcoin-core/reports/20250730/linux/report.html

Info

Unfortunately, the structured fuzzing approach proposed in [MR#26975] did not get approved by the Bitcoin community as it was adding an external dependency on `libprotobuf-mutator` [21].

Additionally, third-party community initiatives — notably `bitcoin-fuzz` ([22]) — aim to find flaws using fuzzing, and specifically differential fuzzing. The Bitcoin project's fuzzing effort is mature - its 221+ harnesses cover a large portion of the codebase. Remaining gaps are in complex components (for example, block connection) and highly structured inputs such as scripts.[13] Section 6 examines current fuzzing effectiveness and offers recommendations for improving coverage and adding new harnesses.

[13] https://github.com/bitcoin/bitcoin/issues/23105

5. Manual Code Review

Bitcoin Core was first released in 2009 and has since evolved over more than a decade of continuous development. To date, the project has accumulated over 45,000 commits from more than 1,000 contributors, although only a small subset remains actively involved in maintaining the codebase. The current codebase consists of approximately 1,454 C/C++ source files, totaling more than 200,000 lines of code (LoC), excluding comments. A summary of the file distribution and line counts is presented in the table below.

Language	files	blank	comment	code
C++	806	28966	28244	184605
C/C++ Header	624	14163	27868	63210
Python	349	13868	14021	57915
Markdown	238	8207	38	31618
C	24	1314	1429	28586
Total	**2041**	**66518**	**71600**	**365934**

Table 3: Summary of the file distribution and line counts

As shown in the table, the repository also includes several auxiliary utilities primarily used for testing and other development-related tasks. Most of the utilities are implemented in Python. Within the C/C++ codebase, a substantial portion is likewise devoted to testing and is located in the `src/test` directory.

An initial code discovery phase was conducted to better understand the overall structure of the codebase and its internal components. The results of this analysis are summarized in the following section.

5.1. Code Discovery

5.1.1. Program & Dependencies

The Bitcoin Core project consists of multiple programs, each serving distinct purposes. These programs are designed to interoperate seamlessly while sharing common components and libraries. The main executables are:

- `bitcoind`: The core Bitcoin daemon providing full-node functionality, including transaction and block validation, peer-to-peer network communication, and consensus management.
- `bitcoin-qt`: The graphical user interface (GUI) built on top of Bitcoin Core, offering a user-friendly environment for interacting with the Bitcoin network and managing wallets.
- `bitcoin-cli`: The command-line interface (CLI) utility that enables direct interaction with the Bitcoin daemon, supporting operations such as sending transactions, querying blockchain data, and managing wallet functionality.

The codebase can also be compiled in a multiprocess configuration as an alternative to the standard multithreaded mode. In this configuration, `bitcoind` is renamed to `bitcoin-node`, while `bitcoin-qt` becomes `bitcoin-gui`. Additionally, the wallet binary called `bitcoin-wallet` works both with or without multiprocessing enabled.

Other utility programs are provided to perform specific tasks without requiring a running node:

- `bitcoin-tx`: A command-line tool for creating and manipulating Bitcoin transactions.
- `bitcoin-util`: A collection of utility functions. At the time of writing, its only feature is computing the proof-of-work (mining) target for a given hexadecimal block header.
- `bitcoin-chainstate`: A tool for inspecting blockchain state by directly manipulating the `datadir` (data directory).
- `libbitcoinkernel`: An upcoming shared library designed to expose Bitcoin functionality to external applications.

Additional binaries are available for testing and benchmarking purposes:

- `test_bitcoin`: A test binary containing all unit tests.
- `test_bitcoin-qt`: A test binary containing GUI tests.
- `bench_bitcoin`: A benchmarking tool for measuring the performance of various components
- `fuzz`: The binary holding all fuzzing harnesses.

Beyond these core tools, Bitcoin Core also provides several optional features that can be enabled at compile time to support specific use cases:

- `External Signer`: Adds support for external hardware or software signing devices.
- `ZeroMQ`: Enables event publishing over a `ZeroMQ` socket for integration with third-party applications.
- `IPC`: Provides inter-process communication (IPC) to support multiprocessing configurations.
- `USDT` (User Defined Tracepoints): Exposes transaction and block events to third-party applications via eBPF hooks [23].
- `DBus`: (Linux only) Enables system notifications via the DBus message bus [24].

These features are intended for specialized scenarios and are disabled by default in standard builds, as they are not required for typical node operation.

5.1.2. Dependencies

The project relies on a number of dependencies, grouped into two categories - external dependencies and internalized dependencies that are directly included in the source tree.

External dependencies:

- Boost 1.73.0
- libevent 2.1.8

Optional external dependencies:

- SQlite3 3.7.17 *(for wallet support)*
- ZeroMQ 4.0 *(for ZeroMQ support)*
- Qt 6.2 *(for GUI support)*

- Cap'n Proto 0.7.1 (multiprocessing mode)

Internalized dependencies:

- univalue for JSON support
- secp256k1 v0.6.0 for cryptographic operations developed in a separate repository [25]
- leveldb 1.22.0 for key-value storage forked from Google LevelDB
- minisketch for set reconciliation, is also developed in a separate repository

Info

While internalized dependencies are not all up to date with the latest upstream versions, none of the updates address security issues. Consequently, synchronizing with newer versions is not considered mandatory.

The project can be compiled using either GCC (version > 11.1) or and Clang (version > 16). The doc/dependencies.md file provides an up-to-date list of dependencies and their respective versions.

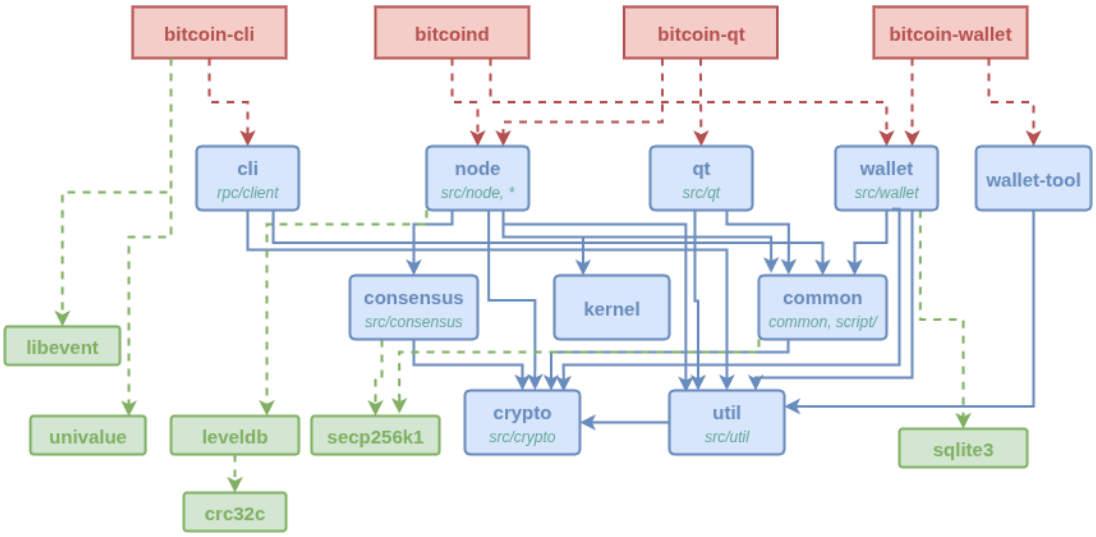


Figure 1: Program and components overview

Figure 1, summarizes the main components of the project and their approximate dependencies - red represents primary binaries, blue denotes core components, and green corresponds to third-party libraries.

5.2. P2P Interface

The peer-to-peer (P2P) interface is the network layer responsible for managing peer connections and propagating data across the Bitcoin network. It is primarily implemented in `src/net_processing.cpp` and `src/net.cpp`. A chain identifier ensures that nodes operate on the correct network (mainnet, testnet, regtest, etc.) and prevents cross-network communication.

From a modular perspective, Figure 2 illustrates the various classes involved in the P2P layer. The cornerstone classes are `CConnman`, which manages connections, and `CNode`, which represents an active peer connection. The protocol currently supports two transport methods, the legacy cleartext `V1Transport` and the newer `V2Transport` defined in BIP-324 [26], which provides end-to-end (E2E) forward-secure encryption. Bitcoin supports both IPv4 and IPv6 connections, as well as Tor and I2P, with these privacy networks implemented *via* configurable proxies within the code.

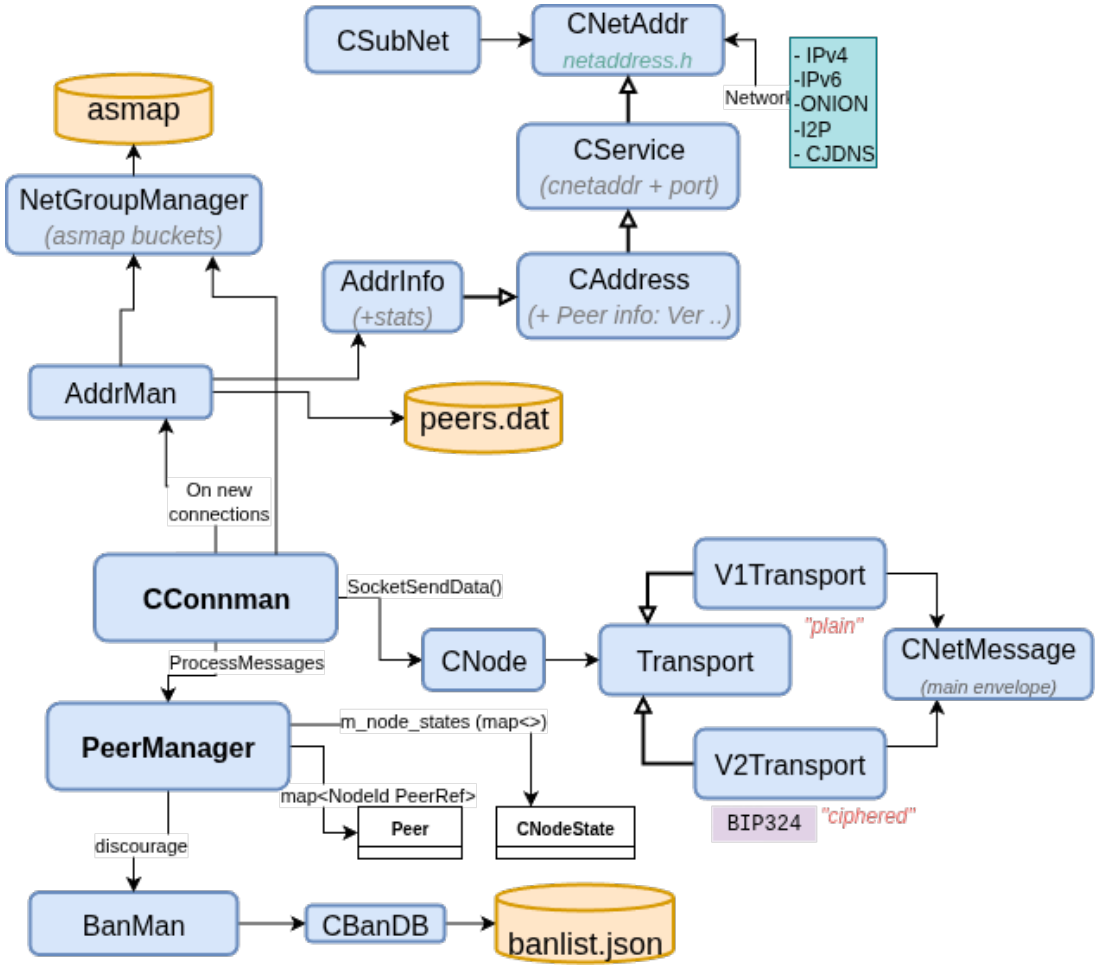


Figure 2: P2P C++ classes

The `PeerManager` class is responsible for maintaining peer information and enforcing a policy of regular behavior for connected nodes. In case of misbehavior, the `BanMan` component can temporarily or permanently ban a peer. This mechanism ensures that flooding or data poisoning attempts are detected and mitigated promptly.

Success

No scenarios were identified that would allow spurious data to be submitted undetected or the ban mechanism to be bypassed. All malicious packets are in a general manner discarded as early as possible in the processing chain.

Finally, the `AddrMan` class is responsible for maintaining a list of known peers and their addresses. A node continuously discovers new peers to ensure connectivity and reduce the risk of **Eclipse** [27] or **Erebus** [28] attacks. To mitigate these attacks, an ASMAP [29] mechanism is implemented, which ensures that new connections are made to peers from different Autonomous System (AS), limiting the risk of network partitioning. This feature is not enabled by default[14]. The `bitcoind` daemon must be started with the `-asmap=<file>` option to activate it. While this mechanism demonstrates the careful attention Bitcoin Core developers have given to network robustness against BGP hijacking, it is not possible to verify whether all peers in the network are using ASMAP.

Because new node addresses are propagated by peers, a node cannot determine *a priori* whether they are valid. The main risk is poisoning the address list with invalid entries. To mitigate this, the implementation discourages or bans nodes that submit invalid addresses. The most recent Bitcoin CVE (CVE-2024-52919) demonstrated that a node could be taken down (DoS) by submitting a large number of addresses, resulting in an integer overflow.

5.2.1. P2P Messages

The `protocol.h` file contains the enum `NetMsgType` defining messages that can be exchanged between nodes through P2P. The different messages are given in Table 4.

[14]However there is ongoing discussion to enable it by default. This raises the question of how to ship asmap files.

Message	Tx	Rx	Description	Category	Introduction
VERSION	•	•	Protocol version info	Handshake	v0.1.0
VERACK	•	•	Acknowledge VERSION	Handshake	v0.1.0
WTXIDRELAY	•	•	WTXID relay support	Node Params	v0.20.0
SENDTXRCNCL	•	•	Tx reconciliation	Node Params	v0.21.0
SENDADDRV2	•	•	Request ADDRV2 support	Node Params	v0.21.0
SENDHEADERS	•	•	Request header updates	Node Params	v0.12.0
SENDCMPCT	•	•	Compact block support	Node Params	v0.13.0
GETADDR	•	•	Request peer addresses	Network discovery	v0.1.0
ADDR	•	•	Address of peers	Network discovery	v0.1.0
ADDRV2	•	•	Extended ADDR message	Network discovery	v0.21.0
INV	•	•	Inventory of objects	Data propagation	v0.1.0
GETDATA	•	•	Request data	Data propagation	v0.1.0
MERKLEBLOCK	•	•	Filtered block	Data propagation	v0.9.0
TX	•	•	Transaction data	Data propagation	v0.1.0
GETBLOCKS	•	•	Request block headers	Data propagation	v0.1.0
BLOCK	•	•	Block data	Data propagation	v0.1.0
CMPCTBLOCK	•	•	Compact block data	Data propagation	v0.13.0
GETBLOCKTXN	•	•	Request block txns	Data propagation	v0.13.0
BLOCKTXN	•	•	Block transactions	Data propagation	v0.13.0
GETHEADERS	•	•	Request headers	Data propagation	v0.1.0
HEADERS	•	•	Block headers	Data propagation	v0.1.0
PING	•	•	Keep-alive message	Connection health	v0.1.0
PONG	•	•	Response to PING	Connection health	v0.1.0
NOTFOUND	•	•	Data not found	Error handling	v0.1.0
MEMPOOL		•	Request mempool data	Mempool sharing	v0.7.0
FILTERLOAD	•	•	Load bloom filter	SPV support	v0.8.0
FILTERADD	•	•	Add to bloom filter	SPV support	v0.8.0
FILTERCLEAR	•	•	Clear bloom filter	SPV support	v0.8.0
GETCFILTERS	•	•	Request compact filters	SPV support	v0.17.0
GETCFHEADERS	•	•	Request CF headers	SPV support	v0.17.0
GETCFCHECKPT	•	•	Request CF checkpoints	SPV support	v0.17.0
FEEFILTER	•	•	Fee rate filter	SPV support	v0.13.0
CFILTER	•		Compact filter data	SPV client	v0.17.0
CFHEADERS	•		Compact filter headers	SPV client	v0.17.0
CFCHECKPT	•		Compact filter checks	SPV client	v0.17.0

Table 4: P2P message types

The Bitcoin protocol does not formally define a state machine as a BIP, but certain messages must follow a specific order. For any message to be accepted, the handshake must be completed first, which requires exchanging `VERSION` and `VERACK` messages. The logic is implemented in

`PeerManagerImpl::ProcessMessage` within the `net_processing.cpp` file, which serves as the main entry point of any incoming message.

Between `VERSION` and `VERACK`, additional signaling messages may be sent to indicate support for various features, such as `WTXIDRELAY`, `SENDHEADERS`, `SENDCMPCT` or `SENDADDRV2`. If these messages are received out of order, they are simply ignored.

After the handshake, all messages are processed normally and dispatched to their respective handlers. Note that certain message types like `CFILTER`, `CFHEADERS` or `CFCHECKPT`, have no handler in a full Bitcoin node because they are only used by SPV (Simplified Payment Verification) clients. If a full node receives them, it simply ignores them.

Success

The various handlers have been reviewed manually, and no instance of misbehavior was identified.

5.2.2. Low-level P2P

At a lower level, messages between nodes are encoded in cleartext using `V1Transport` or encrypted using `V2Transport`.

The cleartext transport is straightforward. Each message consists of a header followed by the message payload. The header includes chain-specific magic value, the message type, the size of the payload, and a the cyclic redundancy check (CRC) checksum of the data. No state management is required beyond tracking partially sent or received message.

The encrypted transport, introduced by BIP-324 [26], is more complex and includes an initialization step. During this phase, peers exchange public keys and synchronize by sending random data combined with the result of the Elliptic-curve Diffie-Hellman (ECDH) operation on the exchanged keys. Before transmitting the first application message, each node sends a version packet internal to the transport protocol (distinct from the standard `VERSION` signaling message). This packet is currently empty but may be used for transport-layer negotiation in future versions. Once initialization is completed, application messages can be sent.

Both the version packet and application messages are encrypted in a same manner. The first three bytes of the encrypted message indicate the length of the ciphered and authenticated payload. The first byte of the decoded message is a header with reserved bits, followed by the message type (either a single non-null byte identifier or a null byte followed by the message type name). The remaining bytes contain the message payload.

If a message authentication and encryption fails, the connection is immediately closed. In contrast, cleartext transport does not terminate the connection when a CRC check fails. Instead, the corrupted message is simply ignored.

Success

No issue have been identified as part of the review process.

5.3. Transaction Validation

Transactions are either received directly into mined blocks or received as to be-mined transactions through the mempool. The Bitcoin Core implementation distinguishes two types of rules that will be applied to transactions:

- **consensus rules**: set of criterias and conditions a transaction should satisfy from a consensus perspective
- **policy rules**: set of additional conditions applied on a transaction for it to be accepted into the mempool

As such, a transaction can be valid from a consensus perspective but rejected from entering the mempool. Policy rules are strictly more restrictive than consensus rules [30]. As such, transactions received in the mempool undergo more checks than those received directly in blocks.

5.3.1. Mempool

The mempool (short for "memory pool") in Bitcoin is the **temporary holding area where unconfirmed transactions reside before being included in a block by miners**. Each full node maintains its own mempool, storing valid transactions that have been broadcast to the network but not yet confirmed on the blockchain. The mempool plays a critical role in transaction propagation, fee prioritization, and network efficiency - nodes relay transactions from their mempool to peers, while miners select transactions from their mempools to maximize block rewards by prioritizing higher-fee transactions. The mempool is not globally uniform, different nodes may have slightly different contents depending on connectivity and policy rules. In essence, the mempool is the staging ground of the Bitcoin network, balancing throughput, security, and economic incentives (fees) until transactions get included in a valid block.

5.3.1.1. Transaction Acceptance Rules

Upon reception via the P2P network, a transaction is processed through a series of functions. If all validation checks succeed, the transaction is added to the mempool. Figure 3 illustrates the call path from `ProcessMessage`. A package is a set of transactions received together and checked sequentially. Transaction acceptance is represented by the `TxValidationResult` which encodes the possible reasons for rejection.

```
enum class TxValidationResult {
    TX_RESULT_UNSET = 0,     //!< initial value. Tx has not yet been rejected
    TX_CONSENSUS,            //!< invalid by consensus rules
    TX_INPUTS_NOT_STANDARD,   //!< inputs (covered by txid) failed policy rules
    TX_NOT_STANDARD,          //!< otherwise didn't meet our local policy rules
    TX_MISSING_INPUTS,        //!< transaction was missing some of its inputs
    TX_PREMATURE_SPEND,       //!< transaction spends a coinbase too early, or violates locktime/
    sequence locks
    TX_WITNESS_MUTATED,  // Transaction might have a witness prior to SegWit activation, or witness may
    have been malleated
    TX_WITNESS_STRIPPED, // Transaction is missing a witness.
    TX_CONFLICT, // Tx already in mempool or conflicts with a tx in the chain
    TX_MEMPOOL_POLICY,        //!< violated mempool's fee/size/descendant/RBF/etc limits
    TX_NO_MEMPOOL,            //!< this node does not have a mempool so can't validate the transaction
    TX_RECONSIDERABLE,        //!< fails some policy, but might be acceptable if submitted in a
    (different) package
    TX_UNKNOWN,               //!< transaction was not validated because package failed
};
```

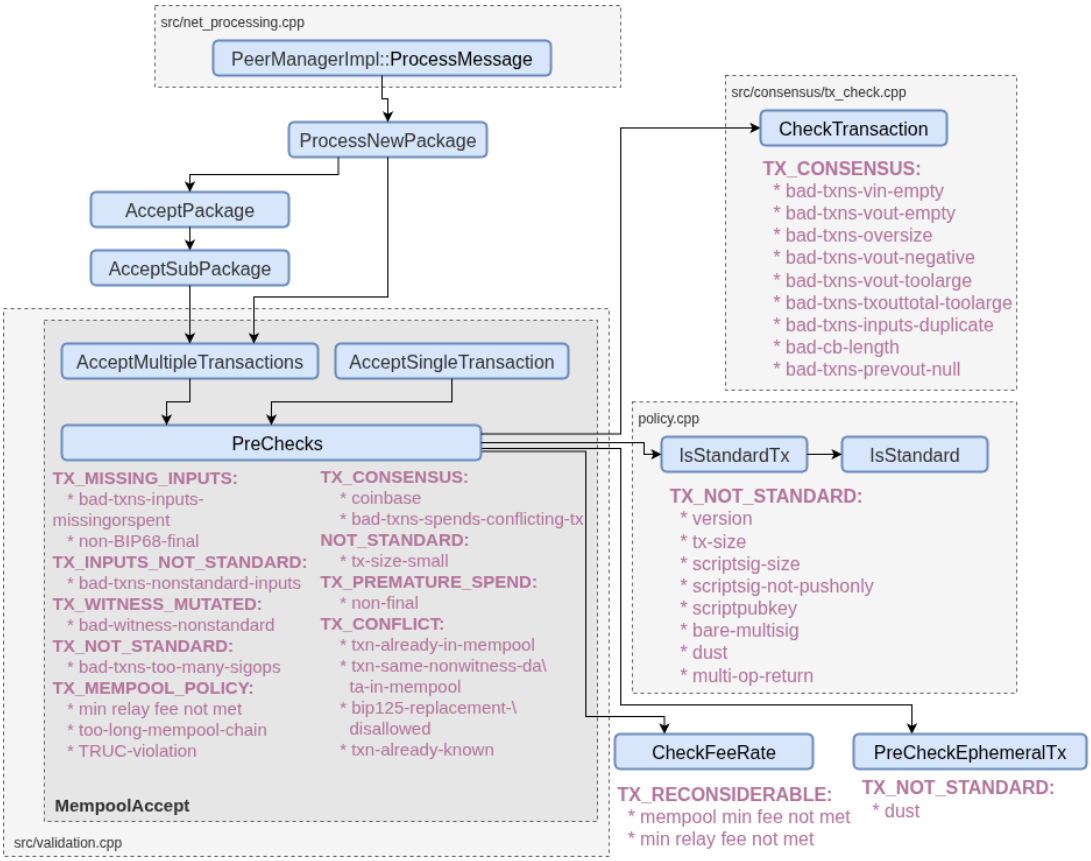


Figure 3: Transaction validation flow (with rejection criterias)

As shown in the figure, `CheckTransaction` is responsible for all consensus-level checks, while `PreChecks` and `IsStandardTx` handle policy-level validations. These rules ensure that only valid, properly signed, and non-conflicting transactions are stored and propagated, preventing spam, double-spends, and other potential abuse such as transaction pinning [31]. For example,

although a coinbase transaction is valid as the first transaction in a block, it cannot be accepted into the mempool.

Success

Static analysis did not reveal any attack scenarios in which an invalid transaction could be accepted into the mempool.

5.3.2. Static Analysis

The static analysis of the clustered-mempool logic was conducted using the `ProcessNewPackage` function as the primary entry point for P2P messages. Within the available time frame, particular attention was given to potential resource exhaustion vectors. Several threat models were defined, including scenarios in which a malicious peer might attempt to overwhelm a node by consuming excessive memory through unbounded package acceptance or by exhausting CPU resources via computationally expensive validation paths. These hypotheses were systematically evaluated against the implementation. The analysis confirmed that the codebase incorporates appropriate safeguards, including strict package size limits, ancestor and descendant constraints, and validation short-circuits that prevent excessive resource usage. Edge cases and boundary conditions were also examined and found to be correctly handled, thereby no attack vectors targeting the mempool were identified.

5.4. Chain Management

From a code perspective, the chain management component is centered on the `ChainStateManager` class, which encapsulates the logic for maintaining the active chain as a `ChainState` instance. The manager maintains two `ChainState` instances: one for the chain being synchronized during initial block download (IBD), and another, called the snapshot, which ultimately represents the canonical chain state. During the synchronization phase, the active pointer references the IBD chain, once the synchronization is complete, it is permanently switched to the snapshot chain. Figure 4 provides an overview of the various modules involved in chain management.

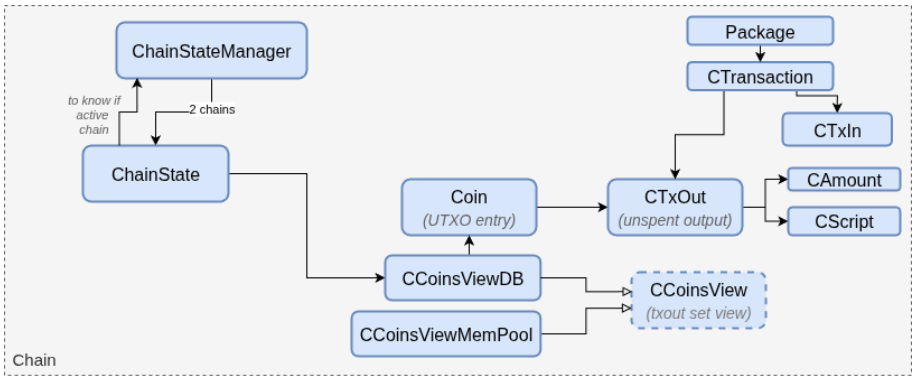


Figure 4: Chain module overview

A chain state contains a `CCoinsViewDB` which provides a view of the UTXO set. This set consists of Coin objects representing the unspent outputs of transactions, each encapsulated in a `CtxOut` object that contains an amount and the associated unlocking script. The UTXO set includes only unspent outputs. Inputs scripts are stored in blocks on disk and are not part of this set.

5.5. Chain Reorganization

Bitcoin's chain reorganization (also referred to as reorg) mechanism allows nodes to maintain consensus when competing chains temporarily diverge. When a node receives a valid chain that has more cumulative proof-of-work than its current active chain, it performs a reorganization - the node disconnects blocks from the tip of its current chain and applies blocks from the new, longer chain.

From a security perspective, reorgs introduce both operational and risk considerations. While short reorgs are normal and expected due to network latency, unusually long reorgs could disrupt transaction finality. Bitcoin mitigates these risks through strict consensus rules, chain work comparison, and mempool management during reorgs, ensuring that transactions are either re-included in the mempool or confirmed in the reorganized chain. Additionally, the protocol's reliance on proof-of-work makes long reorgs economically impractical for an attacker, thereby limiting the feasibility of sustained chain manipulation. This ensures that all nodes converge on the chain with the greatest accumulated work as enforced by consensus rules.

Edge cases, such as simultaneous block arrivals or forks of equal length, are also accounted for by the protocol's deterministic tie-breaking rules, preventing consensus divergence across honest nodes. Overall, the chain reorganization feature is a fundamental component of Bitcoin's resilience, balancing decentralization, consistency, and security.

Blocks are submitted by miners, so it requires more computational power to perform any deleterious actions with blocks as they have to exhibit a valid proof-of-work to be added to the tip of the chain. However, from a threat modeling perspective, malicious miners scenarios are also considered. In this case the practical cost of such an attack can also be evaluated.

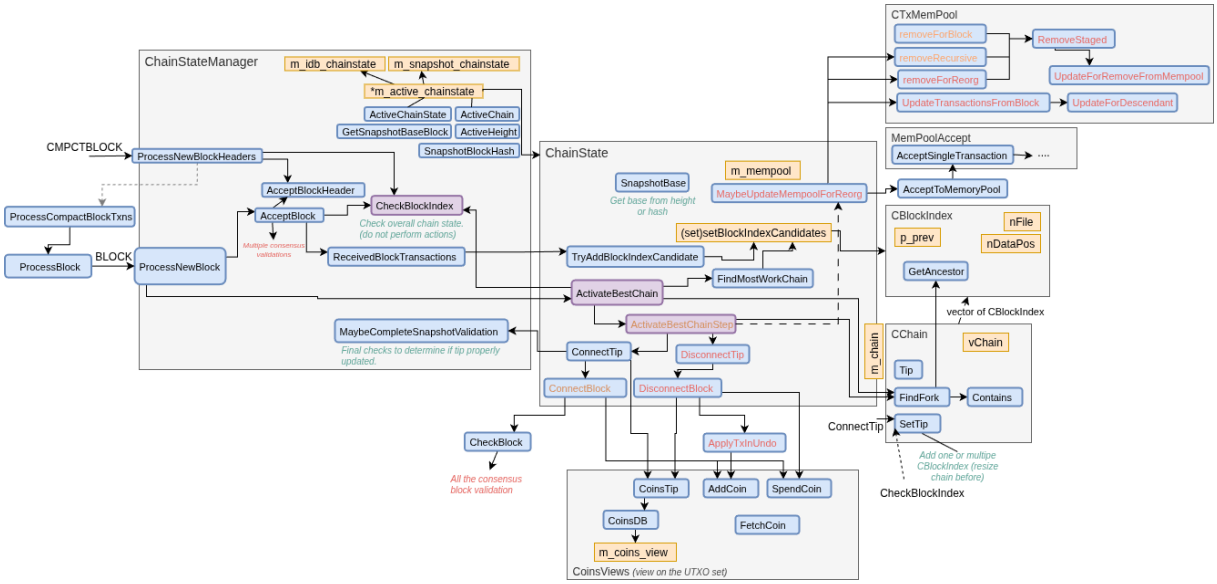


Figure 5: Chain state manager call path

Figure 5 illustrates the call path taken by a block received over P2P. The entry point differs depending on whether compact block mode is enabled. If compact blocks are enabled, `ProcessNewBlockHeaders` is called, otherwise, `ProcessNewBlock` is used. Both functions subsequently invoke `AcceptBlockHeader` and `CheckBlockIndex`[15] to perform consensus checks on the block header. The new tip update is handled within `ChainState` by `ActivateBestChain`, which identifies the chain with the most accumulated work to determine the new tip. If the most-worked chain differs from the current tip, a reorganization occurs using `ConnectTip`, `DisconnectTip`, and `MaybeUpdateMempoolForReorg` to update the tip and adjust the mempool by removing and restoring transactions as needed. All blockchain metadata is maintained in the `CChain` object, which is a vector of `CBlockIndex` entries. Each entry contains the block hash, its location on disk, and a pointer to its parent block index. The actual block content is not stored in memory but is persisted on disk through the `BlockManager` class.

5.5.1. Static Analysis

The static analysis of Bitcoin's chain reorganization (reorg) mechanism focused on its impact on consensus integrity and transaction finality. Auditors examined how nodes handle the arrival of a longer chain, including the disconnection of blocks from the current tip and the application of blocks from the new chain. Several threat models were considered, such as attempts to perform double-spends, censor transactions, or induce denial-of-service through repeated or invalid reorganizations.

Success

[15]This function only exists for sanity checking the block index. It is not enabled on mainnet.

The evaluation confirmed that the protocol enforces strict validation, deterministic tie-breaking, and proper mempool handling during reorgs, effectively mitigating the risk of inconsistencies or consensus divergence.

Overall, the chain reorganization feature is robust, ensuring network convergence while maintaining resistance to economically and computationally infeasible attacks.

Warning

Reorganization-related code is not covered by fuzz testing. As shown in Figure 5, red boxes indicate functions that are not covered at all by fuzz testing. To complement static review, these functions will be addressed in fuzzing phase by implementing a dedicated harness.

5.6. Thread

Threads are employed throughout the project to separate and manage different activities. The following threads have been identified within the codebase:

ThreadName	Description
Main thread	The primary thread that runs the main function. Once initialization is complete, it waits for the shutdown signal
Scheduler thread	Executes scheduled tasks at regular intervals
InitLoad thread	Loads blocks and transactions from disk during startup
Mapport thread	Manages NAT-PMP and PCP protocols to open ports when running behind a NAT and to retrieve the public address
Net thread	Accepts incoming connections on the listening port and receives incoming packets from all sockets. Outgoing packets may also be sent by this thread if not sent directly by the originating thread
DNS seed thread	Discovers new peers during startup using DNS seeds
AddCon thread	Creates and retries outgoing connections controlled by the `-addnode` option or added through RPC
OpenConn thread	Establishes and maintains outgoing connections to other peers
MsgHand thread	Processes incoming messages received from the `Net` thread
I2P Accept thread	Accepts incoming connections from the I2P network
Tor Control thread	Manages connections through the Tor network
Run Command threads	Executes external commands, such as system notifications or hooks
Index threads	Updates various indexes, including the block filter index, coin statistics index, and transaction index
HTTP worker threads	Handles HTTP and RPC requests

Table 5: Threads and their roles

Info

In the above table, some threads are only started if the corresponding command-line argument is provided.

5.6.1. Thread Safety Management

The project leverages Clang's Thread Safety Analysis [32] feature to detect potential issues related to concurrent access of class members. However, this analysis has several limitations that restrict when warnings are generated:

- Only annotated class members are checked. Any variables not explicitly marked are ignored by the analysis.

- The analysis does not appear to track pointers to annotated members. Consequently, if a pointer to a guarded member is returned or used without acquiring the appropriate lock, no warning may be raised.
- The analysis cannot detect deadlocks that occur between multiple threads.

Our review primarily focused on the first limitation by manually reviewing classes and identifying members not protected by thread-safety annotations. Particular attention was given to non-constant, non-atomic variables that are not explicitly guarded.

The assessment of threads and mutexes hightlighted multiple variables that are not **explicitely** marked with `GUARDED_BY` even though, manual analysis confirmed they are rightfully accessed the appropriate mutex lock.

The consequence is that the LLVM *Thread Safety Analysis* is not analyzing them and will not produce any warning if a future code change access them without the appropriate mutex.

The variables are the following:

- `m_tried_collisions` and `key` in `AddrManImpl` should be marked with `GUARDED_BY(cs)`.
- `m_max_addnode` and `m_max_feeler` in `CConnman` should be annotated with `constexp`.
- In `node::BlockManager`, the following variables and functions should be marked either with `GUARDED_BY(::cs_main)` or with `EXCLUSIVE_LOCKS_REQUIRED(::cs_main)`:
  - `m_blockfile_info`
  - `m_blockfile_info`
  - `m_check_for_pruning`
  - `m_dirty_fileinfo`
  - `m_dirty_blockindex`
  - `m_snapshot_height`
  - `m_blocks_unlinked`
  - `FlushBlockFile()`
  - `FlushUndoFile()`
  - `FindNextBlockPos()`
  - `FlushChainstateBlockFile()`
  - `FindUndoPos()`
  - `GetBlockFileInfo()`
  - `WriteBlock()`
  - `UpdateBlockInfo()`
  - `CalculateCurrentUsage()`
- In `Chainstate`, the following variables and functions should be marked either with `GUARDED_BY(::cs_main)` or with `EXCLUSIVE_LOCKS_REQUIRED(::cs_main)`:
  - `m_mempool`
  - `m_coins_views`
  - `m_coinsdb_cache_size_bytes`
  - `m_coinstip_cache_size_bytes`
  - `InitCoinsDB()`
  - `ResetCoinsViews()`
  - `HasCoinsViews()`

  - `PruneAndFlush()`
- In `Chainstate`, the variable `m_chain` is a public member and is sometimes returned by reference. We recommend to implement a read/write mutex for `CChain::vChain`, or at minimum, protect the `m_chain` using the `::cs_main` mutex.
- In `ChainstateManager`, the following variables should be marked by `GUARDED_BY(GetMutex())`:
  - `nBlockReverseSequenceId`
  - `nLastPreciousChainwork`
  - `m_failed_blocks`
  - `m_total_coinstip_cache`
  - `m_total_coinsdb_cache`
- `m_recon_version` in `TxReconciliationTracker::Impl` should be annotated with `const` or guarded by `GUARDED_BY(m_txreconciliation_mutex)`.

INFO-1	Add `GUARDED_BY` attribute to additionnal members
Perimeter	ThreadSafety

Description

Some class members and methods are not annotated with a `GUARDED_BY` mutex, even though they are always accessed while the corresponding mutex is held.

Recommendation

Add `GUARDED_BY` for non-atomic, non-constant class members that are already implicitly covered by a mutex.

LOW-2	Use of `PT_GUARDED_BY` instead of `GUARDED_BY` in `CBlockPolicyEstimator`
Likelihood	●○○○ **Impact** ●○○○
Perimeter	ThreadSafety

Description

Three members (`feeStats`, `shortStats` and `longStats`) in `CBlockPolicyEstimator` are safeguarded with `PT_GUARDED_BY`. In the function `CBlockPolicyEstimator::estimateRawFee()`, the value of these members can be taken before acquiring the lock. If a thread begins executing `CBlockPolicyEstimator::estimateRawFee()` while a second thread holds the mutex while executing `CBlockPolicyEstimator::Read()`, these pointers may become invalid by the time the first thread acquires the mutex.

Recommendation

The `feeStats`, `shortStats` and `longStats` members of `CBlockPolicyEstimator` should be protected with `GUARDED_BY`. The lock of `m_cs_fee_estimator` in `CBlockPolicyEstimator::estimateRawFee()` must be moved to the beginning of the method.

5.7. Automated Static analysis

The automated static analysis was performed using Semgrep [33] with the default community C and C++ rule sets [34].

Success

The output of Semgrep was reviewed manually and yielded only a small number of warnings, none of which were relevant to the security of the codebase.

Info

The use of Semgrep with the default community rules can be integrated into the project's Continuous Integration (CI) pipeline. To remove the existing noise, the warnings could
be filtered out using manual rules.

5.8. Assertion Checking

Bitcoin Core implements a series of sanity checks whose behavior varies depending on whether the code is compiled in `DEBUG` or `RELEASE` mode. Because triggering an assertion failure can terminate the process and thus lead to a potential denial-of-service (DoS) condition, most of these checks are disabled in `RELEASE` builds. Only critical assertions remain active to help preserve runtime stability.

Assertion handling has been the source of several issues in the past, including CVE-2024-35202 and CVE-2024-52919. The following table summarizes the primary macros and functions used for assertions, along with their respective usage contexts.

Macro/Function	DEBUG	RELEASE	Count
CHECK_NONFATAL()	error msg	error msg	183
Assert()	abort	abort	167
Assume()	abort	-	411

The key distinction between the various assertion macros lies in the behavior of `Assume()`, which is active only in `DEBUG` mode. Consequently, these checks can be triggered and observed

during testing or debugging but are entirely omitted in `RELEASE` builds, remaining undetectable at runtime.

As shown in the table above, `Assume()` is primarily used for **property-based testing** by encoding invariants that should always hold true. The `CHECK_NONFATAL()` macro is mainly used within RPC-related functions. These assertions were briefly reviewed through static analysis, with no major issues identified - except for the potentially unsafe use of `Assume()` in place of `Assert()`, as described below.

LOW-3	Risky usage of `Assume` to check null division		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	Fee Calculation		
Prerequisites			

Description

The fee rate computation uses a division function defined as:

```
static inline int64_t Div(__int128 n, int32_t d, bool round_down) noexcept
{
    Assume(d > 0);
    // Compute the division.
    int64_t quot = n / d;
    int32_t mod = n % d;
    // Correct result if the / operator above rounded in the wrong direction.
    return quot + ((mod > 0) - (mod && round_down));
}
```

The function verifies that the divisor `d` is greater than zero using an `Assume()` statement. However, since `Assume()` is only active in `DEBUG` mode, this check is omitted in `RELEASE` builds. Consequently, if the function is invoked with `d` equal to zero, it would lead to a division by zero, resulting in an undefined behavior and a runtime exception. *Note: A review of the possible execution paths leading to this function suggests that a call with $d = 0$ is not currently feasible*

Recommendation

It is recommended to replace `Assume()` with an `Assert()` to ensure that this critical check is performed in both `DEBUG` and `RELEASE` modes.

6. Fuzzing Harnesses

6.1. Introduction

This section evaluates the existing fuzzing harnesses in terms of code coverage and test case relevance. Preliminary assessments have already been conducted by independent researchers [35]. As introduced in Section 4.7.2, Bitcoin Core benefits from an extensive fuzzing framework comprising more than 221 dedicated harnesses. The exhaustive list of harnesses is provided in Appendix C., along with the detailed breakdown of the baseline coverage obtained by running the test cases from the `qa-assets` repository. This repository serves as the canonical source of fuzzing corpora for each harness and is periodically updated to integrate new test cases generated by OSS-Fuzz and to remove obsolete ones, using the merge feature of libFuzzer for corpus optimization. The Bitcoin Core security team relies on OSS-Fuzz [16] for continuous large-scale fuzzing and automated test generation. The resulting coverage metrics can be explored globally on the OSS-Fuzz introspector dashboard or on a per-harness basis[16].

All coverage metrics presented hereafter are generated using LLVM Coverage (and not GCov) as LLVM provides more reliable and consistent results. From the complete set of fuzzing harnesses, a subset was selected for deeper evaluation based on the threat model defined in Section 4.4. Among the key harnesses analyzed are `process_message`, which handles the reception of P2P messages, `process_messages`, which processes multiple messages per iteration, and `script_interpreter`, responsible for executing Bitcoin Scripts.

Bitcoin Core Approach

The Bitcoin Core development team favors an exploratory fuzzing approach, leaving harnesses the duty to autonomously discovering the application's state space, rather than seeding them handcrafted test-cases to start with. This approach avoids the complexity and manual effort required to produce valid inputs for the `FuzzDataProvider`. Over-time, this strategy has naturally yielded a rich and rather comprehensive and exhaustive corpus of test cases.

The next sections aim to evaluate and enhance existing harnesses, as well as to introduce new ones, targeting previously uncovered code regions (see Section 8).

6.2. Detailed Coverage State

A substantial portion of the Bitcoin Core code base is already covered by existing fuzzing harnesses. Some harnesses operate in a manner similar to unit tests, focusing on isolated components, while others aim to reproduce more complex, system-level scenarios that simulate the behavior of a running node. Additionally, a few harnesses such as `kitchen_sink` are designed

[16] http://bitcoind-fuzz.dergoegge.de:8000/bitcoin/harnesses/ (link available at 2025-11-01)

primarily to extend coverage artificially by targeting otherwise unreachable code paths. As a result, the overall baseline coverage is relatively high. However, several areas remain uncovered:

Tor: The Tor integration code remains partially uncovered by fuzzing harness. Solely the `torcontrol` harness addresses it. Fuzzing this component offers limited value due to its external dependencies and relatively small attack surface. Its behavior would be better assessed through integration or end-to-end testing rather than fuzzing.

Block validation: Core block validation logic implemented in *validation.cpp*, still contains uncovered regions, particularly around chainstate initialization and data flushing to disk. Of greater concern, the function `MaybeUpdateMempoolForReorg` is not covered. The routine is invoked during a chain reorganization, indicating that no existing harness has successfully triggered such a scenario. Consequently, dependent functions such as `ApplyTxInUndo`, `DisconnectBlock`, and `DisconnectTip` are also never executed during fuzzing, meaning no block was ever disconnected from the active chain tip. These gaps will be specifically addressed by the new harness described in Section 8.4. Similarly, some error-handling branches in `ConnectBlock` remain untested and will be targeted by the dedicated harness described in Section 8.3.

Transaction validation: is primarily handled by the `MemPoolAccept` class and its associated functions. Within `MemPoolAccept::PreChecks()`, the case corresponding to `txn-already-known` is not covered. This condition occurs when a transaction attempts to create a `COut` already known in the `CoinCache`, a scenario that existing harnesses should theoretically be able to produce.

Additionally, in the `CalculatePrevHeights` routine, which retrieves input `Coins` for transactions entering the mempool, the else-case for coin retrieval is never reached. This implies that no fuzz-generated transactions contain invalid inputs, such as:

1. Missing UTXO: The referenced output does not exist in the UTXO set.
2. Already spent UTXO: The output has already been consumed by another transaction.
3. Invalid outpoint: The referenced transaction ID or output index is malformed.

Moreover, within *net_processing.cpp*, the `ProcessMessage` function, responsible for handling incoming network messages, shows that, for messages of type `NetMsgType::TX`, no valid transaction was ever successfully generated or processed by the fuzzing harness, as illustrated in Figure 6.

```
13.7k        if (result.m_result_type == MempoolAcceptResult::ResultType::VALID) {
    0            ProcessValidTx(pfrom.GetId(), ptx, result.m_replaced_transactions);
    0            pfrom.m_last_tx_time = GetTime<std::chrono::seconds>();
    0        }
13.7k        if (state.IsInvalid()) {
```

Figure 6: Coverage in ProcessMessage function for NetMsgType::TX case

It is possible that valid transactions were generated by other harnesses, but not by process_message itself.

6.3. Relevant Harnesses Selection

Among the 221 existing harnesses, a subset specifically targets P2P components that are critical from a security standpoint. Following discussions with the Bitcoin Core team, a list of harnesses was selected for a more detailed assessment:

Harness Name	Description
`script_flags`	Read a transaction and associated flags in order to call `VerifyScript` (do not used FuzzDataProvider)
`utxo_total_supply`	Test adequation between UTXO total supply and blocks mined by miner
`process_messages`	Add multiple peers receive multiple messages and process them
`coinscache_sim`	Exercise the coins cache by adding, spending, querying it
`script_interpreter`	Test SignatureHash function on transaction read from input
`eval_script`	Test `EvalScript` function with a CScript read from input
`coins_view`	Exercise the coins view by adding, spending, querying it
`process_message`	Test P2P message reception
`tx_pool`	Test transaction mempool acceptance
`ephemeral_package_eval`	Test ephemeral package evaluation (RBF etc)
`tx_package_eval`	Test Package evaluation on mempool
`txdownloadman`	Test tx reception, new tip, etc.. (complex init function!)
`txdownloadman_impl`	Test TxDownloadManagerImpl
`txorphan`	Test orphanage features

For these harnesses, an evaluation of corpus relevance was conducted, and potential avenues for improvement were explored. Overall, they achieve broad and comprehensive coverage, with only a few minor unexplored areas remaining. The harness `utxo_total_supply` was observed to execute relatively slowly (approximately 304 executions per second), primarily due to its repeated initialization steps required to create a valid chainstate at each iteration. Unless otherwise noted in the following sections, no specific issues or concerns were identified in these harnesses.

6.4. Assessing `process_message` harness

The `process_message` harness tests the reception of a single P2P message and is therefore among the most relevant targets from a threat-modeling perspective. It constructs artificial peer connections and simulates an incoming message from one of them. The harness exercises `PeerManagerImpl::ProcessMessage`, the central dispatcher for handling incoming P2P messages. The current corpus contains inputs for all message types, however, some message flows, such as `CMPTBLOCK`, require a prior request to be sent before the node will process the subsequent message. Because `process_message` covers only a single message reception, it cannot exercise such multi-message exchanges.

A major challenge for this harness is the heterogeneity of inputs across message types (see Table 4). Fortunately, the resulting fuzzing corpus is reasonably balanced across message types and captures much of the per-type coverage potential. As expected, `tx` and `inv` messages dominate the corpus. Figure 7, summarizes the message-type distribution in the corpus.

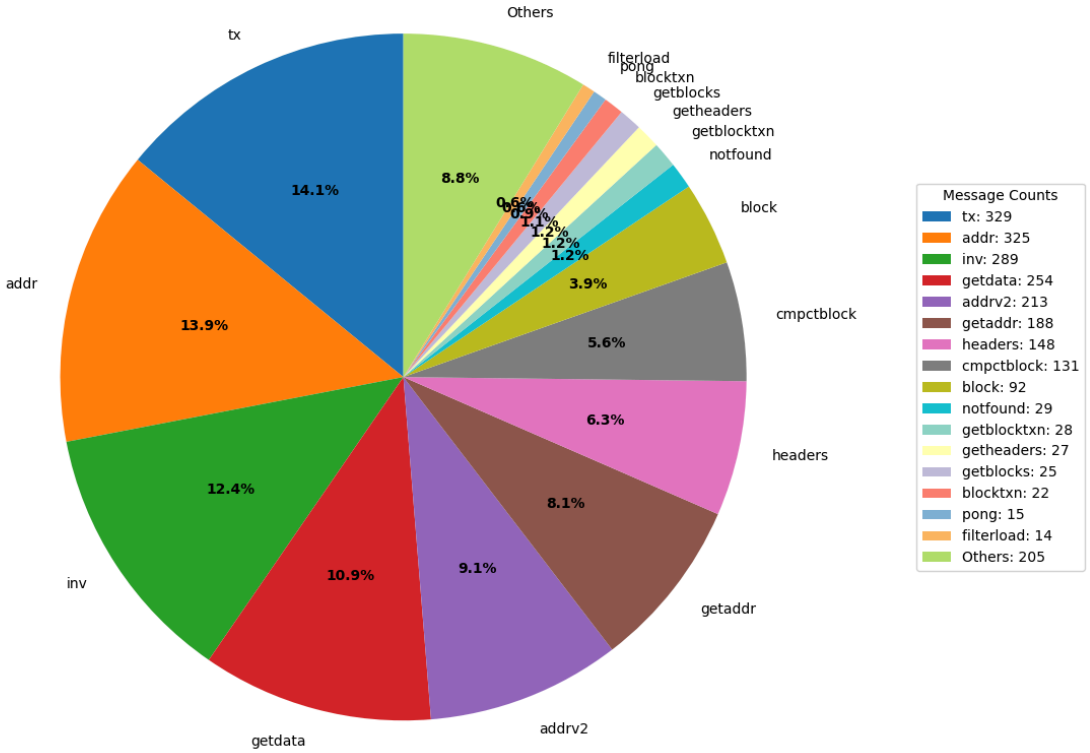


Figure 7: Message types distribution in corpus

While this harness, can in theory generate transactions and blocks, in practice the search space is too wide. As shown in Figure 8 no transaction can validate all the `PreChecks` (see Figure 3).

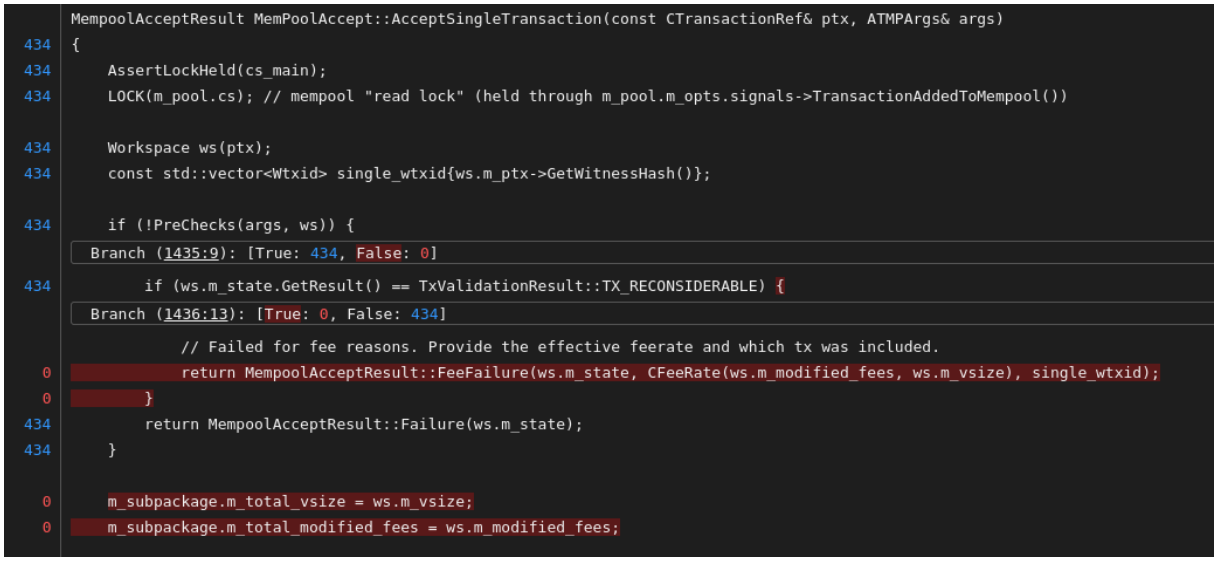


Figure 8: `AcceptSingleTransaction` *coverage*

This harness and few various other suffer from a limitation on the initial state. The initialization function[17] pre-mine blocks so that multiple coinbases become spendable.

```
    for (int i = 0; i < 2 * COINBASE_MATURITY; i++) {
        MineBlock(g_setup->m_node, {});
    }
```

While any "user" can spend these coinbases, nothing is made at the harness level to help the fuzzer to meet the requirements to spend these coinbases.

INFO-4	No pre-determined or hardcoded key materials to spend coins
Perimeter	Fuzzing Harnesses
Prerequisites	

Description

Most harnesses rely solely on the mutation capabilities of libFuzzer to generate valid data. This includes, transaction or blocks. This likely slows down or hinder the exploration capabilities of the fuzzer.

Recommendation

It is recommend to help fuzzing by casually providing it materials like keys, coins known *a priori*, that can be used by the harness to generate valid transactions or blocks. Another option is to generate coins that do not requires any keys to be spent.

[17] called once when fuzzing

6.5. Assessing `script_sigcache`

The harness invokes either `VerifySchnorrSignature` or `VerifyECDSASignature`, providing a public key, a signature (as `bytes`), and a hash (as a `uint256`) to verify the signature against. These three values are directly read from the fuzzing input. In practice, it is virtually impossible for a fuzzer to generate a valid signature corresponding to a given public key and hash, as doing so would require knowledge of the associated private key. Consequently, the practical relevance of this harness is limited. Nevertheless, analyzing the resulting coverage still provides valuable insights.

```
    bool CachingTransactionSignatureChecker::VerifyECDSASignature(const std::vector<unsigned char>& vchSig, const CPubKey& pubkey, const uint256& sighash) const
125 {
125     uint256 entry;
125     m_signature_cache.ComputeEntryECDSA(entry, sighash, vchSig, pubkey);
125     if (m_signature_cache.Get(entry, !store))
  0         return true;
125     if (!TransactionSignatureChecker::VerifyECDSASignature(vchSig, pubkey, sighash))
122         return false;
  3     if (store)
  1         m_signature_cache.Set(entry);
  3     return true;
125 }

    bool CachingTransactionSignatureChecker::VerifySchnorrSignature(std::span<const unsigned char> sig, const XOnlyPubKey& pubkey, const uint256& sighash) const
237 {
237     uint256 entry;
237     m_signature_cache.ComputeEntrySchnorr(entry, sighash, sig, pubkey);
237     if (m_signature_cache.Get(entry, !store)) return true;
237     if (!TransactionSignatureChecker::VerifySchnorrSignature(sig, pubkey, sighash)) return false;
  0     if (store) m_signature_cache.Set(entry);
  0     return true;
237 }
```

Figure 9: Coverage of `sigcache.cpp`

As expected, and as shown in Figure 9, no valid Schnorr signatures were generated during fuzzing. Surprisingly, however, some valid ECDSA signatures were successfully produced. A closer examination of these valid test cases reveals the following field values:

```
vchSig:
3002020102020102416 3c80000006ca81757147d630a00312d3232d532ffff74ba000000c8fffff90000\
        0000000000000000000000000000000000000000000000000000000000000000
pubkey: 020000000000000000000000000000000000000000000000000000000000000002
sighash: 0000000000000000000000000000000000000000000000000000000000000000
```

The implementation attempts to sign the hash value 0 using a public key composed of 33 bytes set to `0x00`, except for the last byte set to `0x02`[18]. Out of 471 generated test cases, three resulted in valid ECDSA signatures[19]. All of them follow the same pattern.

For these three test cases, the signatures are considered valid by the `secp256k1` library but invalid by OpenSSL under the same conditions. The discrepancy arises from the use of `ecdsa_signature_parse_der_lax`, which parses DER-encoded signatures in an overly permissive manner by not verifying the declared sequence length. This lenient behavior is intentionally preserved for backward compatibility.

[18] The leading 0x02 indicates a compressed public key format.

[19] af66aa89b5b12a3b4164843f55cceea3140b70d5, b0bdd09dcf5ab01ca754337cb6417f7d88d744fd, and d2a8c81fa23bea1eae4f52313e6df2c3c1149e77

BIP-66 introduced stricter DER encoding rules for ECDSA signatures with a soft fork. Accordingly, Bitcoin Core enforces these constraints through the `IsValidSignatureEncoding` function, invoked during transaction validation within the Bitcoin Script interpreter. The fuzzing harness triggers this discrepancy because it interfaces directly with `VerifyECDSASignature` and `VerifySchnorrSignature`, bypassing the higher-level validation layer. Consequently, the issue cannot be exploited in practice. Appendix F. provides a minimal code snippet reproducing this behavior using the parameters discovered through fuzzing.

Warning

The function `ecdsa_signature_parse_der_lax` is overly permissive in its parsing logic and should not be used as-is without additional validation checks. The source file explicitly warns about the security risks associated with using this function.

6.6. Other harnesses

All remaining harnesses were also briefly reviewed to identify potential design or implementation issues. This section summarizes the key improvements that could be applied to enhance their robustness and maintainability.

INFO-5	`bip324_cipher_roundtrip`: fuzz highly entropic bytes
Perimeter	Fuzzing Harness
Prerequisites	

Description

The harness reads two private keys and an entropy seed from the input. While functional, embedding high-entropy bytes directly in the fuzz input forces the fuzzer into a near-pure brute-force search. Consequently, much mutation effort is wasted on randomizing those entropy bytes without meaningful progress in exploring program behavior.

Recommendation

Avoid reading high-entropy bytes directly from the fuzz input. Because private keys and seeds are effectively random, embedding them wastes mutational effort. They should instead be be drawn from a secure random source.

INFO-6	Lack of exercising results for multiple harnesses
Perimeter	Fuzzing Harness

Prerequisites

Description

Several harnesses execute operations but neither validate nor otherwise exercise the results, which limits their usefulness for fuzzing. When the output of an operation is ignored, the fuzzer loses an important feedback signal and faulty behaviors can be missed. For example, the `blockmerkleroot` parses a `CBlock` and computes the Merkle root, but the computed value is never inspected or asserted against a reference. Also, `crypto_fschacha20` repeatedly invokes `FSChaCha20.Crypt` but never checks return values or the produced ciphertext.

Recommendation

Ensure results of computations and function calls are validated via assertions or additional checks.

INFO-7	`ellswift_roundtrip`: fuzz highly entropic bytes
Perimeter	Fuzzing Harness
Prerequisites	

Description

Like `bip324_cipher_roundtrip`, the `ellswift_roundtrip` harness reads three fields from the input - a private (random) key, some randomness, and a `uint256` value to sign. None of these fields require specific values to trigger interesting behavior, so mutating them amounts largely result to brute-force exploration. As a result, the fuzzer wastes cycles mutating high-entropy bytes with little chance of producing meaningful state changes.

Recommendation

Avoid reading high-entropy bytes directly from the fuzz input. Because private keys and seeds are effectively random, embedding them wastes mutational effort. They should instead be drawn from a secure random source.

INFO-8	`descriptor_parse`: implementation impedes fuzzing coverage readability
Perimeter	Fuzzing Harness
Prerequisites	

Description

The harness tests descriptor parsing using the following code snippet:

```
for (const bool require_checksum : {true, false}) {
    const auto desc = Parse(descriptor, signing_provider, error, require_checksum);
    std::optional<bool> is_ranged;
    std::optional<bool> is_solvable;
    for (const auto& d : desc) {
        assert(d);
        TestDescriptor(*d, signing_provider, error, is_ranged, is_solvable);
    }
}
```

It exercises `Parse` with `require_checksum` set successively to `true` and `false`. However, the resulting code coverage within this function does not allow direct determination of whether `Parse` succeeded when a checksum was required. Determining that requires analyzing subsequent calls to verify the checksum result. While this limitation does not directly hinder fuzzer discoverability, it complicates result interpretation for analysts. An assessment of the `qa-assets` corpus revealed that only a single test case out of 2224 was able to generate a valid checksum, likely due to sheer luck. Standard fuzzers are generally ineffective at inferring relationships between disparate parts of an input, such as checksums. Further analysis of `Parse` and the underlying `CheckChecksum` also exposed behavior that may be unintended (see details below).

Recommendation

While the harness is written in a very compact form, it is recommended that, for post-fuzzing analysis, the code be made more self-explanatory. In this case, unrolling the for loop into two distinct loops—one with `require_checksum` set to true and the other to false —would improve readability and make the results easier to interpret.

INFO-9	`parse_iso8601`: Cast `int32_t` into `int64_t`
Perimeter	Fuzzing Harness
Prerequisites	

Description

The `parse_iso8601` performs the following actions:

```
const int64_t random_time = fuzzed_data_provider.ConsumeIntegral<int32_t>();
//[..]
const std::string iso8601_datetime = FormatISO8601DateTime(random_time);
```

It reads a 32-bit integer into a 64-bit integer to call `FormatISO8601DateTime`. As a result, the upper 32 bits only contain the sign extension of the original 32-bit value. Since `FormatISO8601DateTime` is designed to handle full 64-bit values, limiting the input to 32 bits may miss certain behaviors, although this does not appear to be an issue in practice.

Recommendation

The harness should directly consume a 64-bit integer.

INFO-10	`primitive_transaction`: Assertions are tautologies
Perimeter	Fuzzing Harness
Prerequisites	

Description

The `primitive_transaction` performs the following actions:

```
const CTxOut tx_out_1{ConsumeMoney(fuzzed_data_provider), script};
const CTxOut tx_out_2{ConsumeMoney(fuzzed_data_provider), ConsumeScript(fuzzed_data_provider)};
assert((tx_out_1 == tx_out_2) != (tx_out_1 != tx_out_2));
```

Given the semantics of the equality and inequality operators, this assertion is always true and can never be violated. A similar assertion is present elsewhere in the harness.

Recommendation

The contents of the parsed transaction should be actively manipulated and tested using meaningful checks rather than relying on tautological assertions.

INFO-11	Using FuzzingDataProvider in `script_flags`
Perimeter	Fuzzing Harness
Prerequisites	

Description

Almost all harnesses are using `FuzzingDataProvider` to consume bytes of the input. The `script_flags` harness is an exception, as it reads its fields using a `DataStream` object instead. In both cases, there are no security implications and no direct impact on fuzzing effectiveness.

Recommendation
Modifying the harness to use `FuzzingDataProvider` in order to be more consistent and uniform with the other harnesses.

INFO-12	`socks5`: field `username`, `password` needlessly too long
Perimeter	Fuzzing Harness
Prerequisites	

Description
In the `socks5` protocol, the `username` and `password` fields are each set to 512 bytes. These fields play only a minor role in the harness.

Recommendation
Reduce both field sizes to facilitate fuzzer mutations and improve exploration.

Note on harnesses

Multiple harnesses, such as `addition_overflow` or `crypto_aes256`, are implemented in a unit-test style. The functions they exercise have a very limited scope, and the harnesses contain almost no branching conditions. As a result, the entire code surface can be covered in a single run, which makes the benefit of coverage-guided fuzzing questionable. In such cases, the fuzzer effectively falls back to state-coverage without meaningful guidance. To improve the usefulness of these harnesses, explicit assertions should be added to verify that incorrect or unexpected values are correctly detected and handled.

Overall, the existing harnesses are carefully designed and well implemented. Some are structured as traditional unit tests, while others encode complex scenarios and action sequences using constructs like `LIMITED_WHILE` and `CallOneOf`. This is both a strength and a limitation - while it enables deep exploration of system behavior, it also forces the fuzzer to navigate a very large state space.

Additionally, the semi-structured interface provided by `FuzzedDataProvider` may not always be sufficient to guide the exploration effectively. For example, the `process_message` harness, as evaluated above, indicates that libFuzzer can reach deep states, but likely at a high computational cost[20]. Some harnesses also operate without input size constraints, with `p2p_transport_serialization` processing inputs up to 1 MB.

[20] endorsed by OSS-Fuzz

Finally, several harnesses consume pseudo-random bytes directly from the input. By their nature, fuzzers and their evolutionary algorithms are unlikely to generate meaningful values from these high-entropy fields. These considerations are summarized in the findings below.

INFO-13	Reduce `LIMITED_WHILE` iterations
Perimeter	Fuzzing Harness
Prerequisites	

Description

Several harnesses, such as `process_message` and `policy_estimator`, allow up to 10,000 loop iterations per call. This number appears excessively high, as the full range of possible behaviors can typically be explored with far fewer iterations. While memory consumption is unlikely to be problematic at this scale, the additional iterations provide limited benefit relative to the computational cost.

Recommendation

Reducing the number of iterations to a more reasonable value, for example to 1000.

INFO-14	Limit input size for fuzzing
Perimeter	Fuzzing Harness
Prerequisites	

Description

The `qa-asset` reference corpus includes some very large inputs, reaching several megabytes. It is highly unlikely that these oversized inputs provide additional value beyond what smaller, more targeted inputs already achieve.

Recommendation

It is recommended to limit input sizes to a reasonable value (e.g., 8 kB). A harness that requires very large inputs to trigger new behaviors is likely suboptimal, as libFuzzer persistent harnesses are designed to iterate inputs very quickly. This limitation can be further mitigated by regularly merging and trimming existing test cases to maintain efficiency and avoid unnecessary overhead.

INFO-15	Reading highly entropic fields from the input

Perimeter	Fuzzing Harness
Prerequisites	

Description

Some harnesses, particularly those related to cryptography, read key material, such as private keys or random seeds, directly from the input. While it is theoretically possible for some values to trigger interesting behaviors, this approach tends to hinder mutation effectiveness and fuzzer performance. For example, in `secp256k1_ec_seckey_import_export_der`, the values read are entirely random, leaving the fuzzer no feasible way to generate relevant inputs for these fields.

Recommendation

It is recommended to avoid reading highly entropic fields from the input directly or to strongly limit its size.

7. Ensemble Fuzzing

Current fuzzing setup relies on OSS-Fuzz for continuous fuzzing and requires by consequence libFuzzer-compatible harnesses. These constraints prevent from leveraging the diversity of existing fuzzers and their functionalities. While the harness supports AFL++ persistent mode, it is only casually run and not on Google infrastructure. OSS-Fuzz is notoriously known to have the following limitations:

1. No corpus is being shared by the independent instances of libFuzzer
2. It is libFuzzer centric, AFL++ is integrated but not actively maintained and run on harnesses
3. It does not leverage fuzzer specific features like AFL++, COMPCOV-CMPLOG to improve its coverage
4. It runs single instances for a few hours only and then terminates. This CI setting, hinders the possibility to discover deep bugs requiring long campaigns.

To address these points, **ensemble fuzzing** is the mean to combine various fuzzers and at larger extend test engines to leverage their respective capabilities and strengths. Recent works have highlighted some benefits of using various fuzzers [36] or ensemble fuzzing [37]. The main foundation behind ensemble fuzzing is that no fuzzer is universally better than all others on all targets. Indeed, they differ in the way they instrument the target, in the way they mutate inputs, in the way they select inputs to fuzz, etc. It also enables combining different testing approaches like fuzzing, symbolic execution or model checking.

The principal difficulty in fuzzing lays in the harnesses engineering and the input mutation algorithm to use. Ensemble fuzzing only comes at the end as the cherry on the cake. As such, there is only very few frameworks available. Most of them are abandoned [38], [39]. AFL++ and LibAFL provides distributed and seed synchronization modules but are restricted to their own fuzzer. To our knowledge only PASTIS [37] provides a framework to run ensemble fuzzing with heterogeneous fuzzers[21].

7.1. PASTIS

PASTIS[22], is an ensemble fuzzing written in Python integrating AFL++, Honggfuzz [40], libFuzzer and TritonDSE [41]. It can fuzz programs both instrumentated from source, as well as binary-only ones. In this latter case, QEMU mode, and QBDI [42], are respectively used by AFL++ and Honggfuzz.

PASTIS uses ZeroMQ, for all messages exchanges between fuzzers and the main controller called *broker*. As such, the fuzzing can be parallelized and scaled across multiple machines. The main role of the broker is the input synchronization between fuzzers. As every engine might implement different coverage mechanism and metrics the only artefacts shared between engines are inputs. This approach offers the best flexibility to integrate new fuzzers. AFL++ supports

[21] and not only multiple instances of the same fuzzer.
[22] https://quarkslab.github.io/pastis/index.html

synchronization with a foreign queue (`-F`) option. We added the support for Honggfuzz[23] while libFuzzer is restarted periodically to take new inputs in account.

The broker can be run in 3 modes:

- no synchronization, the broker aggregates fuzzer's coverage running independently
- full synchronization, all inputs are sent to all fuzzers (maximal sharing approach)
- full synchronization with filtering, only inputs generating new coverage are transmitted to other fuzzers

Previous studies have shown that the maximal sharing can hinder the performances of fuzzing engines thus campaigns will be run with filtering enabled. For it to work the broker replays all inputs received on the target compiled with LLVM Coverage instrumentation. The coverage generated is compared with the current one. If new coverage is found the input is sent to all other fuzzers.

Coverage Synchronization

libFuzzer uses Sanitizer Coverage (SanCov) to compute its coverage. The implementation tends to keep significantly more inputs than LLVM-Cov as they intend to approach state coverage rather than simple branch coverage. These inputs do not necessarily generates new effective coverage in LLVM Coverage. As such, using LLVM-Cov will reject multiple SanCov input and thus reduce the number of inputs shared. This restrictive approach ensures only inputs truly covering new artifacts are shared and ensure them to be beneficial for other engines.

7.2. Preparing the `fuzz` binary

To run ensemble fuzzing, one need to build the `fuzz` binary with the instrumentation specific to each fuzzer. Five variants are generated for AFL++, Honggfuzz, libFuzzer and LLVM Coverage for the final coverage computation. An additional AFL++ variant is compiled with `CMPLOG` to leverage the efficient comparison feature of AFL++ *(implementing Redqueen [43])*. Table 6, summarizes the exact compilation flags used for each variants. Note that Honggfuzz and AFL++ were compiled without AddressSanitizer. Indeed, there is no need to compile all variants with AddressSanitizer.

[23] https://github.com/google/honggfuzz/pull/500

Fuzzer	Compilation options
libFuzzer	`cmake -B build_libfuzzer -DCMAKE_C_COMPILER="clang" -DCMAKE_CXX_COMPILER="clang++" -DCMAKE_C_FLAGS="-ftrivial-auto-var-init=pattern" -DCMAKE_CXX_FLAGS="-ftrivial-auto-var-init=pattern" -DSANITIZERS="undefined,address,fuzzer" -DBUILD_FOR_FUZZING=ON`
AFL++	`cmake -B build_aflpp -DCMAKE_C_COMPILER="afl-clang-lto" -DCMAKE_CXX_COMPILER="afl-clang-lto++" -DBUILD_FOR_FUZZING=ON -DSANITIZERS=address,undefined` *(and another with `export AFL_LLVM_CMPLOG=1`)*
Honggfuzz	`cmake -B build_hfuzz -DCMAKE_C_COMPILER="hfuzz-clang" -DCMAKE_CXX_COMPILER="hfuzz-clang++" -DBUILD_FOR_FUZZING=ON`
LLVM Coverage	`cmake -B build_cov_llvm_fuzz -DCMAKE_C_COMPILER="clang" -DCMAKE_CXX_COMPILER="clang++" -DAPPEND_CFLAGS="-fprofile-instr-generate -fcoverage-mapping -fcoverage-mcdc" -DAPPEND_CXXFLAGS="-fprofile-instr-generate -fcoverage-mapping -fcoverage-mcdc" -DAPPEND_LDFLAGS="-fprofile-instr-generate -fcoverage-mapping -fcoverage-mcdc" -DBUILD_FOR_FUZZING=ON`

Table 6: Compilation option used for each fuzzers

No dictionary was used for the campaign as the one generated by AFL++ is done on the whole program instead of the 215+ harnesses independently. Tokens from one harness are consequently irrelevant for another one. No time was spent sorting them out.

7.3. Fuzzing Campaign

Campaigns were run using a dev branch of PASTIS[24] on two different servers. Based on their importance, harnesses were run from 20minutes up to 24h for the most important ones. The whole campaigns did span over several weeks. The initial corpus used was taken from **qa-assets**[25] from April 11, 2025.

7.3.1. Coverage Results

Table 7 shows the resulting LLVM Coverage improvements against the initial corpus baseline.

[24] https://github.com/quarkslab/pastis/tree/refactor-coverage-replay
[25] https://github.com/bitcoin-core/qa-assets

Harness	Funcs	Insts	Region	Lines	Branches	MCDC
address_deserialize	208	318	687	1082	290 (+1)	14
base32_encode_decode	20	22	95 (+1)	123 (+2)	50 (+5)	12 (+4)
base64_encode_decode	18	20	77 (+1)	108 (+2)	34 (+3)	5 (+2)
build_and_compare_feerate_diagram	32	36	199	192	93 (+1)	15
chacha20_split_crypt	45	51	180 (+3)	457 (+3)	62 (+2)	6
chacha20_split_keystream	36	41	169 (+3)	419 (+3)	65 (+2)	7
coinselection_knapsack	298	346	970 (+2)	1769 (+3)	361 (+1)	35
connman	625	878	3429 (+2)	4544 (+2)	1490 (+2)	199
crypter	430 (+2)	456 (+2)	2853 (+71)	4801 (+85)	707 (+52)	48 (+14)
crypto_aeadchacha20poly1305	64	71	277 (+3)	719 (+3)	105 (+2)	7
crypto_poly1305	33	34	111 (+3)	286 (+3)	37 (+2)	4
crypto_poly1305_split	33	34	132 (+3)	304 (+3)	49 (+2)	4
decode_tx	193	285	540 (+1)	914 (+2)	211 (+3)	23 (+2)
descriptor_parse	824	1055	6090 (+2)	8361	3488 (+2)	390 (+1)
ephemeral_package_eval	821	1154	4392 (+2)	5468 (+2)	1604 (+2)	70
fee_rate	42	47	233 (+2)	277 (+2)	110 (+2)	8
feefrac	34	37	200 (+1)	178	79 (+2)	4 (+2)
flatfile	58	70	233	332	100 (+1)	7 (+2)
hex	246	372	843	1551	316 (+2)	40 (+4)
integer	208	286	1068	1534	475 (+2)	53 (+3)
load_external_block_file	445	705 (+1)	2040 (+8)	2733 (+4)	792 (+2)	43
mini_miner	428	551	1533 (+2)	2292 (+2)	434 (+2)	23
mini_miner_selection	531	709	1933 (+2)	3073 (+2)	564 (+2)	18
miniscript_script	191	298	1647	1852	1207	148 (+1)
miniscript_string	145	165	1424 (+1)	1672	1081 (+1)	100
mocked_descriptor_parse	832	1065	6161	8443	3543	403 (+1)
netaddr_deserialize	170	190	583	903	261 (+1)	12 (+1)
netaddress	283	344	1349 (+5)	2165 (+4)	802 (+6)	110
p2p_handshake	650	871	3460 (+2)	4285 (+2)	1285 (+2)	153
p2p_headers_presync	777	1450	4052 (+2)	4887 (+2)	1289 (+2)	38
parse_numbers	28	40	249	269	165 (+2)	46 (+1)
parse_univalue	608	811	4721 (+1)	6664 (+1)	2840 (+1)	359 (+3)
partially_downloaded_block	470	715	1557 (+2)	2464 (+2)	475 (+2)	26
partially_signed_transaction_deserialize	376 (+6)	856 (+11)	1421 (+31)	2802 (+80)	702 (+25)	55 (+6)
policy_estimator	349 (+10)	581 (+10)	1411 (+140)	2097 (+169)	541 (+88)	34 (+14)
process_message	1261 (-1)	2526 (+1)	8751 (+8)	9966 (+11)	3690 (+5)	396 (-2)
process_messages	1287	2590	9025 (+2)	10259 (+2)	3908 (+3)	459 (+1)
protocol	65	79	271	396	128 (+1)	10 (+1)
psbt	735	1097	4638 (+2)	7338 (+5)	2521 (+3)	312 (+2)
psbt_base64_decode	310 (+6)	506 (+9)	1238 (+27)	2442 (+59)	599 (+17)	55 (+3)
psbt_input_deserialize	336 (+8)	763 (+19)	1123 (+34)	2311 (+89)	487 (+28)	38 (+4)
psbt_output_deserialize	240 (+4)	338 (+8)	827 (+14)	1587 (+36)	324 (+11)	30 (+1)
random	57	63	165 (+3)	353 (+3)	44 (+2)	3
script	520	605	2610	4033	1583 (+2)	195 (+2)
script_flags	496	645	3787	5408	1732 (+1)	247 (+1)
script_ops	117	129	429	697	217 (+1)	36 (+3)
script_sign	802	1115	5595	7909	2726	291 (+1)
service_deserialize	189	221	622	977	268 (+1)	12 (+1)
signature_checker	236	253	1571	2254	1021 (+1)	148 (+1)
span	14	15	48	76	13 (+1)	2
transaction	595	993	2427 (+3)	4061 (+5)	1303 (+2)	122
tx_package_eval	856	1217	4762 (+2)	5940 (+2)	1971 (+2)	118
tx_pool	1154	1566	7362 (+2)	10025 (+2)	3048 (+2)	285
tx_pool_standard	924	1305	4965 (+2)	6361 (+2)	1853 (+2)	77
txdownloadman	483	644	1778 (+2)	2753 (+2)	673 (+2)	56
txdownloadman_impl	475	636	1921 (+2)	2849 (+2)	715 (+2)	58
txgraph	374	408	2716 (+3)	3392 (+5)	1149 (+2)	91
txoutcompressor_deserialize	181	244	711	1247	247 (+1)	34 (+1)
utxo_snapshot_invalid	666	1126	2685	4486	808 (+1)	50
wallet_create_transaction	1296	1699	7679 (+2)	10531 (+2)	2937 (+2)	146
wallet_notifications	1206	1653	6045 (+2)	8936 (+2)	2175 (+2)	108

Table 7: Harness coverage improvements by Ensemble Fuzzing

Funcs are functions covered, **Insts** are template instances, **Regions** correspond roughly to blocks of instructions (basic block), **Lines** are source lines, **Branches** are control-flow branches. Then **MCDC** is the Modified Condition/Decision Coverage that computes the capability inputs tests to covers independently each terms of a branching conditions. Without any changes in harnesses code, *Ensemble Fuzzing* was able to improve the coverage on 60 different harnesses. This confirms the added value of combining multiple fuzzers to *"go the extra mile"* in coverage.

Some harnesses were significantly improved like `crypter`, `partially_signed_transaction_deserialize` or `policy_estimator`. Manually investigating results shows the added value. For instance, Figure 10 and Figure 11 shows coverage improvements brought by the `crypter` harness. In the first figure, fuzzing was able to cover additional branches in the ASN.1 DER signature parsing function `secp256k1_ecdsa_signature_parse_der_lax`. The second figure shows a whole new region covered in the `ChaCha20::Keystream()` function taken in `m_bufleft` is true.

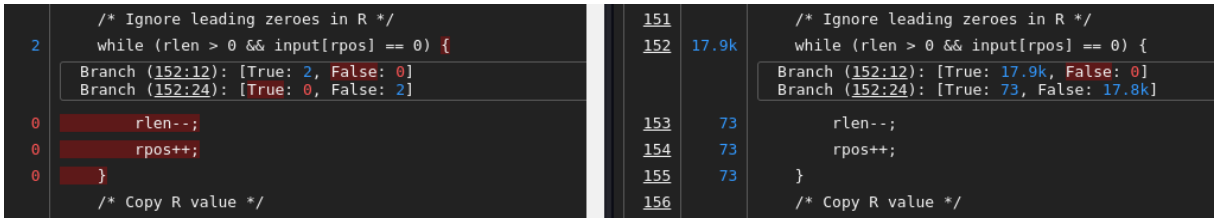


Figure 10: Coverage improvement in `secp256k1_ecdsa_signature_parse_der_lax`

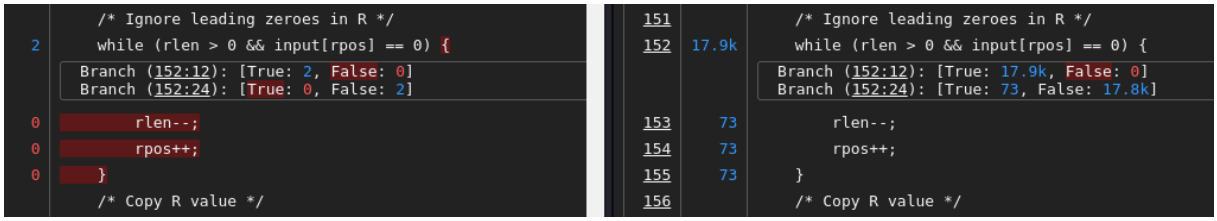


Figure 11: Coverage improvement in `ChaCha20::Keystream()`

In relevant harness like `process_message` few coverage artifacts are also added. On this harness, among the 5 new covered branches one is *script.cpp* as shown in Figure 12.

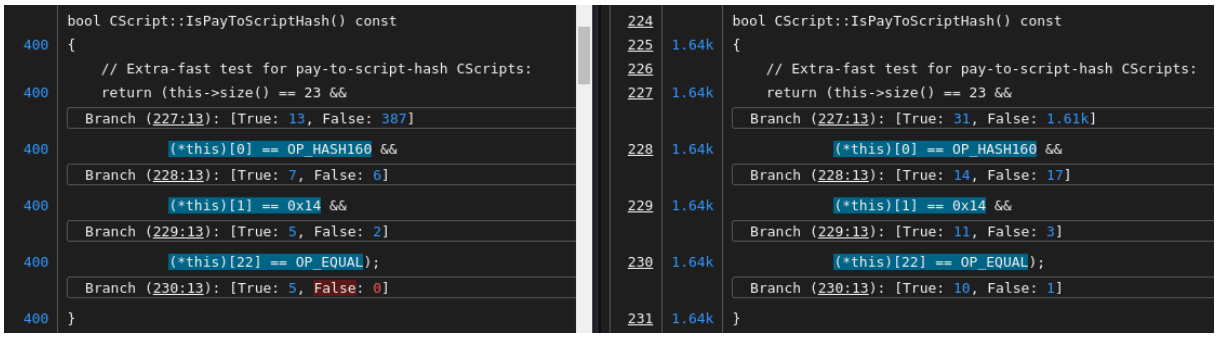


Figure 12: New branch covered in `CScript::IsPayToScriptHash()`

Info

The coverage presented here is computed on a per-harness basis. As such, lines newly covered by a harness might be already covered by another one. The overall coverage improvement is shown in next section. Still, coverage improvements shall be computed individually to determine whether Ensemble Fuzzing brings an added value.

Exact results with detailed lines, branches and MCDC coverage for each harness is provided to Brink's team as supplementary material of the report.

7.3.2. Corpus Results

Harness	Init/Merged	Corpus	Final-Merged	LLVM-Cov
address_deserialize	130/92	454	91 (-1)	33
base32_encode_decode	34/25	210	27 (+2)	8
base64_encode_decode	24/15	154	17 (+2)	6
build_and_compare_feerate_diagram	70/60	442	55 (-5)	9
chacha20_split_crypt	282/48	259	47 (-1)	7
chacha20_split_keystream	357/75	263	68 (-7)	9
coinselection_knapsack	287/152	1513	148 (-4)	23
connman	1697/1044	5746	1036 (-8)	127
crypter	354/88	2048	341 (+253)	52
crypto_aeadchacha20poly1305	395/65	620	61 (-4)	6
crypto_poly1305	29/25	129	22 (-3)	8
crypto_poly1305_split	49/40	419	33 (-7)	6
decode_tx	367/114	738	114	22
descriptor_parse	2224/1247	4144	1234 (-13)	286
ephemeral_package_eval	847/523	3932	489 (-34)	39
fee_rate	13/7	62	9 (+2)	5
feefrac	27/17	97	18 (+1)	11
flatfile	29/25	206	25	9
hex	230/144	552	144	34
integer	333/49	441	47 (-2)	40
load_external_block_file	292/209	861	219 (+10)	17
mini_miner	601/298	1246	285 (-13)	13
mini_miner_selection	498/271	2430	275 (+4)	6
miniscript_script	598/461	1930	462 (+1)	116
miniscript_string	634/451	1122	433 (-18)	100
mocked_descriptor_parse	2527/1504	3111	1491 (-13)	308
netaddr_deserialize	87/67	297	65 (-2)	29
netaddress	389/132	625	128 (-4)	64
p2p_handshake	731/488	3008	457 (-31)	80
p2p_headers_presync	618/238	1601	266 (+28)	14
parse_numbers	108/77	562	70 (-7)	35
parse_univalue	2101/1210	4338	1158 (-52)	309
partially_downloaded_block	479/268	2752	273 (+5)	28
partially_signed_transaction_deserialize	1592/642	6394	644 (+2)	168
policy_estimator	526/130	1916	295 (+165)	67
process_message	2335/1412	7260	1398 (-14)	307
process_messages	3103/2032	1401	2066 (+34)	268
protocol	22/21	164	19 (-2)	11
psbt	3896/1992	4366	1996 (+4)	384
psbt_base64_decode	1224/537	2556	528 (-9)	152
psbt_input_deserialize	737/425	5400	438 (+13)	110
psbt_output_deserialize	365/250	3031	278 (+28)	60
random	832/27	187	24 (-3)	9
script	2119/500	3374	464 (-36)	255
script_flags	2088/1144	4706	1115 (-29)	184
script_ops	182/122	1726	126 (+4)	27
script_sign	3503/1923	7145	1844 (-79)	256
service_deserialize	92/72	324	67 (-5)	30
signature_checker	1897/642	3763	604 (-38)	184
span	4/3	15	3	3
transaction	950/470	2764	467 (-3)	63
tx_package_eval	981/604	3576	583 (-21)	59
tx_pool	2986/1898	6561	1804 (-94)	247
tx_pool_standard	918/497	3079	503 (+6)	37
txdownloadman	498/356	2785	356	19
txdownloadman_impl	528/363	1988	349 (-14)	19
txgraph	2258/576	5715	579 (+3)	46
txoutcompressor_deserialize	65/47	377	55 (+8)	34
utxo_snapshot_invalid	175/110	552	110	36
wallet_create_transaction	1149/735	242	723 (-12)	65
wallet_notifications	936/582	48	573 (-9)	0

Table 8: Harness Input Corpora Improvement

The generated corpus is given in Table 8. The **Init/Merged** column gives the number of initial seed inputs and the number once merged by libFuzzer. The whole initial corpus

was provided for the the fuzzing. **Corpus** gives the aggregated corpus generated by engines in PASTIS. Then **Final/Merged** gives the number of inputs remaining after merging by libFuzzer. Finally **LLVM-COV** gives the number of relevant inputs as computed by LLVM Coverage and which output is visualizable in HTML.

After fuzzing the merged corpus can be smaller than the merged initial one. This is expected, and happens when libFuzzer finds a more concised subset of inputs covering the same or better coverage. Reducing the final corpus is thus positive as a reduced corpus benefits fuzzers.

7.4. Overall Results

7.4.1. Coverage Improvement

Previous results, once merged into a consolidated coverage are presented in Table 9. As there can be coverage overlap between harnesses, numbers are potentially lower than the sum of individual harnesses. Still, 7 new functions were covered and more than 200 lines of code. Results are very satisfying given the coverage saturation generated by the countless hours of fuzzing Bitcoin Core has already received.

	Functions	Instances	Regions	Lines	Branches	MCDC
Base	6202	12637	55307	76211	27028	2827
New	6209	12651	55456	76434	27143	2846
Final	**+7** *(+0.08%)*	**+14** *(+0.05%)*	**+149** *(+0.16%)*	**+223** *(+0.19%)*	**+115** *(+0.26%)*	**+19** *(+0.34%)*

Table 9: Overall LLVM Coverage improvement by Ensemble Fuzzing

Table 10 provides the detailed view of functions which coverage has been improved. The **New** are functions that are newly covered. **Regs** and **Br** are respectively new regions and branches within these functions. One can observe that multiple of them are serialization, deserialization template instances. There are also `TxConfirmStats` which is also not particularly important. More interestingly, new regions, and branches have been covered in `CKey::Sign`, `CKey::VerifyPubKey` and also in `PeerManagerImpl::ProcessMessage`.

Function	New	Regs	Br
unsigned int ReadVarInt<SpanReader, (VarIntMode)0, unsigned int>(SpanReader&)	No	+2	+1
void PSBTInput::Unserialize<DataStream>(DataStream&)	No	+13	+3
void DeserializeMuSig2ParticipantPubkeys<DataStream>(DataStream&, ...)	No	+2	+1
void DeserializeMuSig2ParticipantDataIdentifier<SpanReader>(SpanReader&, CPubKey&, ...)	Yes	+4	+1
void CPubKey::Set<unsigned char*>(unsigned char*, unsigned char*)	No	+2	
void PSBTOutput::Unserialize<DataStream>(DataStream&)	No	+2	+1
void PSBTInput::Serialize<DataStream>(DataStream&)	No	+20	+8
VectorWriter& VectorWriter::operator<<<std::span<unsigned char const, ...	Yes	+1	
void Serialize<VectorWriter, unsigned char const>(VectorWriter&, std::span<...>)	Yes	+1	
void SerializeToVector<DataStream, CompactSizeWriter, ...	Yes	+1	
void SerializeMany<SizeComputer, CompactSizeWriter, ...	Yes	+1	
void SerializeMany<DataStream, CompactSizeWriter, ...	Yes	+1	
void SerializeToVector<DataStream, CompactSizeWriter, ...	Yes	+1	
void SerializeMany<SizeComputer, CompactSizeWriter, ...	Yes	+1	
void SerializeMany<DataStream, CompactSizeWriter, ...	Yes	+1	
void PSBTOutput::Serialize<DataStream>(DataStream&)	No	+4	+2
feefrac.cpp:(anonymous namespace)::MulCompare(long, long, long, long)	No	+1	
miniscript_string_fuzz_target(std::span<unsigned char const, 18446744073709551615ul>)	No	+2	+1
CKey::Sign(uint256 const&, std::vector<unsigned char, std::allocator<unsigned char>>&, bool, unsigned int)	No	+2	+1
CKey::VerifyPubKey(CPubKey const&)	No	+2	+1
CNetAddr::GetReachabilityFrom(CNetAddr const&)	No	+1	+1
wallet::KnapsackSolver(std::vector<wallet::OutputGroup, std::allocator<wallet::OutputGroup>>&, long const&, long, FastRandomContext&, int)	No	+4	
wallet::DecryptKey(std::vector<unsigned char, secure_allocator<unsigned char>> const&, ...)	No	+2	+1
void CKey::Set<__gnu_cxx::__normal_iterator<unsigned char*, std::vector<unsigned char, secure_allocator<unsigned char>>>>(...)	No	+2	
void LogPrintFormatInternal<unsigned long, long>(std::basic_string_view<char, std::char_traits<char>>, ...)	No		
BanMan::DumpBanlist()	No		
net_processing.cpp:(anonymous namespace)::PeerManagerImpl::ProcessBlockAvailability(long)	No		
net_processing.cpp:(anonymous namespace)::PeerManagerImpl::ProcessMessage(CNode&, std::__cxx11::basic_string<char, ...)	No	+2	+1
TxConfirmStats::Record(int, double)	Yes	+8	+1
TxConfirmStats::EstimateMedianVal(int, double, double, unsigned int, EstimationResult*...)	No	+4	+2
TxConfirmStats::NewTx(unsigned int, double)	Yes	+1	
TxConfirmStats::removeTx(unsigned int, unsigned int, unsigned int, bool)	Yes	+43	+9
CBlockPolicyEstimator::_removeTx(uint256 const&, bool)	No	+2	+1
CBlockPolicyEstimator::processTransaction(NewMempoolTransactionInfo const&)	No	+19	+4
CBlockPolicyEstimator::processBlockTx(unsigned int, RemovedMempoolTransactionInfo const&)	No	+14	+1
CBlockPolicyEstimator::processBlock(std::vector<RemovedMempoolTransactionInfo, ...)	No	+11	+2
CBlockPolicyEstimator::estimateRawFee(int, double, FeeEstimateHorizon, EstimationResult*...)	No	+2	+1
CBlockPolicyEstimator::BlockSpan()	No	+7	+1
CBlockPolicyEstimator::estimateCombinedFee(unsigned int, double, bool, EstimationResult*...)	Yes	+29	+7
CBlockPolicyEstimator::estimateConservativeFee(unsigned int, EstimationResult*...)	Yes	+8	+2
CBlockPolicyEstimator::estimateSmartFee(int, FeeCalculation*, bool)	No	+25	+7
CBlockPolicyEstimator::Write(AutoFile&)	No	+2	+1
CBlockPolicyEstimator::FlushUnconfirmed()	No	+2	+1
CBlockPolicyEstimator::TxStatsInfo::TxStatsInfo()	Yes	+1	

Table 10: Newly covered or improved function coverage (unexhaustive list)

File	Functions	Instances	Regions	Lines	Branches	MCDC
src/banman.cpp	15	15	183 (-4)	143	40 (-1)	4
src/key.cpp	25	25	290 (+2)	334 (+3)	112 (+2)	11
src/logging.h	7	79 (-1)	17	22	8	2
src/net_processing.cpp	81	98	2727 (-5)	2340 (-2)	1258 (-4)	155 (-1)
src/netaddress.cpp	73	73	652 (+1)	719 (+1)	573 (+1)	106
src/policy/fees.cpp	36 (+5)	36 (+5)	490 (+124)	563 (+145)	228 (+78)	24 (+9)
src/policy/fees.h	1 (+1)	1 (+1)	1 (+1)	1 (+1)	0	0
src/protocol.cpp	10	10	66	66	43 (+1)	7 (+1)
src/psbt.h	20 (+1)	42 (+3)	474 (+25)	914 (+70)	416 (+27)	17 (+4)
src/script/descriptor.cpp	189	189	1432	1626	907	112 (+1)
src/secp256k1/src/modules/extrakeys/main_impl.h	14	14	171	156	50 (+1)	2 (+1)
src/serialize.h	118	2647 (+5)	236	455	76	3
src/streams.h	61	345 (+1)	192	270	82	5
src/test/fuzz/feefrac.cpp	6	6	246 (+1)	135	130 (+1)	13
src/test/fuzz/feeratediagram.cpp	6	6	93	78	55 (+1)	4
src/test/fuzz/miniscript.cpp	61	64	616 (+1)	672	477 (+1)	48
src/test/fuzz/span.cpp	1	1	3	16	2 (+1)	0
src/util/strencodings.cpp	28	34	271	323	205 (+4)	59 (+4)
src/wallet/coinselection.cpp	37	37	344 (+2)	488 (+3)	222 (+1)	34
src/wallet/crypter.cpp	8	8	52 (+1)	85 (+2)	27 (+1)	1

Table 11: File coverage improvement Ensemble Fuzzing

Table 11 provides the file-wise coverage improvement. The file `src/policy/fees.cpp` has received a lot of improvements. Unexpectedly, `src/net_processing.cpp` and `src/banman.cpp` coverage performed worse than the baseline. A thorough analysis is necessary to explain this phenomenon. But, as PASTIS uses the initial corpus to initiate the campaign, we only expect to get coverage additions.

7.4.2. Corpus Improvement

This section tallies the additional test inputs generated and provided to Bitcoin Core developers to enrich the main **qa-assets** corpus.

One can evaluate the individual contribution of each fuzzer on existing harnesses. Honggfuzz is undeniably the most prolific of the three fuzzers as it submitted 484 that got accepted and transmitted to other fuzzers. On the other hand, libFuzzer and AFL++ only submitted 57 and 13 relevant inputs respectively. AFL++ is somewhat imbalanced as it was only run on a single core while Honggfuzz and libFuzzer were run on 4 cores.

Honggfuzz	libFuzzer	AFL++
484	57	13

Table 12: Input counts contribution to new LLVM Coverage artifacts per fuzzers

From each harness, one can merge the generated corpus so it could later be added in qa-assets. As shown on table 960 new files were added, but −1275 removed. That can be explained by the fact that as inputs cover overlapping areas, libFuzzer might find a way to obtain the exact same coverage with fewer inputs, and thus reports a lower input count.

Added	Removed	Net
+960	−1275	−315

Table 13: Overall libFuzzer Corpus Files

Results obtained have shown that involving multiple fuzzing engines enables generating a beneficial extra coverage leap. PASTIS still had not been able to identify any vulnerability but better covers the codebase probably at lower cost than OSS-Fuzz. Also, while it can be difficult to integrate PASTIS within OSS-Fuzz, it might be beneficial to run it periodically to gather additional tests cases.

Delivrable

As part of this fuzzing campaign, we provide:

- a docker image to run fuzzing campaigns automatically using the main PASTIS framework
- the augmented input corpora that will be merged in qa-assets

8. New Fuzzing Harnesses

8.1. Introduction

Based on the current fuzzing coverage and discussions with the Bitcoin Core developers, several components have been identified as lacking dedicated harnesses. In particular, the block connection and chain reorganization processes are currently not covered, even though harnesses such as `process_messages` could theoretically exercise these paths.

As illustrated previously in Figure 5, various components are not covered and especially chain reorganization logic. No existing harness is capable of generating inputs complex enough to trigger a full chain reorganization.

Info

Previous efforts, had been made to create harnesses for chainstate management. Especially PR29158, introduced `blockstorage` and `chainstate` two harnesses to test chain state and reorganizations. Unfortunately, it has never been merged.

To address this limitation, Quarkslab auditors first implemented a harness for `ConnectBlock`, which allows exercising all consensus-related checks (for both blocks and transactions) without producing side effects on the chain state. Building on this foundation, additional harnesses were developed for `ConnectTip`, `ActivateBestChainStep`, and finally `ActivateBestChain`. The latter two can, in theory, trigger a chain reorganization by selecting the most "worked" chain through the `FindMostWorkChain` function.

8.2. Virtual FileSystem

Block connection and disconnection operations require reading and writing block and undo data to disk via the `BlockManager` class. Since libFuzzer operates in a persistent mode, any files left over from a previous iteration can affect subsequent runs, compromising reproducibility. Therefore, harnesses must carefully clean up any files created during execution. Moreover, minimizing disk I/O is essential, as filesystem access and related system calls significantly slow down iteration cleanup and, consequently, the entire fuzzing campaign. To address these issues, all file operations were designed to run entirely in memory.

To achieve this, we introduced a new abstraction layer for the `File*` object. In Bitcoin Core, most file objects are opened using the `fsbridge::fopen` function. We modified this behavior by adding a fuzzer-specific mode that redirects such calls to `fopencookie`. Our implementation of `fopencookie` provides similar functionality to `open_memstream`, with the additional ability to truncate files and expand file size without reallocating the entire buffer. This behavior is encapsulated in a dedicated class named `MemoryFile`.

In addition to modifying the `open` behavior, the `fsbridge` interface has been extended with the following functions:

```
namespace fsbridge {
    using FopenFn = std::function<FILE*(const fs::path&, const char*)>;
    FILE *fopen(const fs::path& p, const char *mode);
    fs::path AbsPathJoin(const fs::path& base, const fs::path& path);

    /* Added */
    bool isMemoryFile(FILE* f);
    bool truncateMemoryFile(FILE* f, unsigned int length);
    void setMemFSDatadir(const fs::path& p);
    void setEnableMemFS(bool v);
    bool isPathInMemFS(const fs::path& p);
    bool shouldBeMemoryFile(const fs::path& p);
    bool createSnapshotMemFS();
    bool restoreSnapshotMemFS();
    bool clearMemFS();
    /* End Added */

    class FileLock
    {
      // [...] unchanged
    };
};
```

The additional functions introduced enable full management of the virtual filesystem directly from the harnesses. Beyond handling individual files, we implemented a filesystem abstraction named `MemoryFS`, which allows reopening files previously written to, by referencing their path. This abstraction can also be snapshotted and restored to a prior state using the functions `createSnapshotMemFS` and `restoreSnapshotMemFS`.

Info

In some cases, the target functions were not publicly accessible members of their classes or were declared as static. To allow the creation of harnesses for such functions, certain declarations were modified to expose the target functions to the harness code.

8.3. Block connection #1 with `ConnectBlock`

The `ConnectBlock` function plays a critical role in ensuring that received blocks comply with consensus rules before being integrated into the blockchain. To achieve this, it invokes both the `CheckBlock` and `CheckTxInputs` functions. Under normal operating conditions, none of the `CheckBlock` verifications are expected to fail, as these checks are already performed earlier in the call chain within `AcceptBlock`. Table 14 summarizes the various checks executed by `ConnectBlock` along with their respective coverage status.

Function Call Tree	Enum	Code	Cov	Description
AcceptBlockHeader	BLOCK_CACHED_INVALID	duplicate-invalid	✓	block cached as invalid (the reason is not stored)
↳ CheckBlockHeader	BLOCK_INVALID_HEADER	high-hash	✓	proof of work failed
AcceptBlockHeader	BLOCK_MISSING_PREV	prev-blk-not-found	✓	We don't have the previous block the checked one is built on
AcceptBlockHeader	BLOCK_INVALID_PREV	bad-prevblk	✓	A block this one builds on is invalid
↳ ContextualCheckBlockHeader	BLOCK_INVALID_HEADER	bad-diffbits	✓	incorrect proof of work
↳ ContextualCheckBlockHeader	BLOCK_INVALID_HEADER	time-too-old	✓	block's timestamp is too early
↳ ContextualCheckBlockHeader	BLOCK_INVALID_HEADER	time-timewarp-attack	✓	block's timestamp is too early on diff adjustment block
↳ ContextualCheckBlockHeader	BLOCK_TIME_FUTURE	time-too-new	✓	block timestamp too far in the future
↳ ContextualCheckBlockHeader	BLOCK_INVALID_HEADER	bad-version(XX)	✓	reject blocks with outdated version
AcceptBlockHeader	BLOCK_HEADER_LOW_WORK	too-little-chainwork	✓	the block header may be on a too-little-work chain
ConnectBlock				
↳ CheckBlock				
| ↳ CheckBlockHeader	BLOCK_INVALID_HEADER	high-hash	✓	proof of work failed
| ↳ CheckMerkleRoot	BLOCK_MUTATED	bad-txnmrklroot	✓	hashMerkleRoot mismatch
| ↳ CheckMerkleRoot	BLOCK_MUTATED	bad-txns-duplicate	✓	Check for merkle tree malleability
↳ CheckBlock	BLOCK_CONSENSUS	bad-signet-blksig	×	signet block signature validation failure
↳ CheckBlock	BLOCK_CONSENSUS	bad-blk-length	✓	size limits failed
↳ CheckBlock	BLOCK_CONSENSUS	bad-cb-missing	✓	first tx is not coinbase
↳ CheckBlock	BLOCK_CONSENSUS	bad-cb-multiple	✓	more than one coinbase
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-vin-empty	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-vout-empty	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-oversize	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-vout-negative	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-vout-toolarge	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-txouttotal-toolarge	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-inputs-duplicate	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-cb-length	✓	-
| ↳ CheckTransaction	TX_CONSENSUS	bad-txns-prevout-null	✓	-
↳ CheckBlock	BLOCK_CONSENSUS	bad-blk-sigops	✓	out-of-bounds SigOpCount
ConnectBlock	BLOCK_CONSENSUS	bad-txns-BIP30	×	tried to overwrite transaction (BIP30 violation)
↳ CheckTxInputs	TX_MISSING_INPUTS	bad-txns-inputs-missingorspent	✓	inputs missing/spent (double spend)
↳ CheckTxInputs	TX_PREMATURE_SPEND	bad-txns-premature-spend-of-coinbase	✓	tx spends a coinbase too early, or violates locktime
↳ CheckTxInputs	TX_CONSENSUS	bad-txns-inputvalues-outofrange	✓	-
↳ CheckTxInputs	TX_CONSENSUS	bad-txns-in-belowout	✓	value in (%s) < value out (%s)
↳ CheckTxInputs	TX_CONSENSUS	bad-txns-fee-outofrange	×	bad transaction fee
ConnectBlock	BLOCK_CONSENSUS	bad-txns-accumulated-fee-outofrange	×	accumulated fee in the block out of range
ConnectBlock	BLOCK_CONSENSUS	bad-txns-nonfinal	×	contains a non-BIP68-final transaction
ConnectBlock	BLOCK_CONSENSUS	bad-blk-sigops	×	too many sigops
↳ CheckInputScripts	TX_NOT_STANDARD	non-mandatory-script-verify-flag	✓	Didn't meet our local policy rules
↳ CheckInputScripts	TX_CONSENSUS	mandatory-script-verify-flag-failed	✓	
ConnectBlock	BLOCK_CONSENSUS	bad-cb-amount	×	coinbase pays too much
ConnectBlock	BLOCK_CONSENSUS	mandatory-script-verify-flag-failed	×	

Table 14: Checks and associated error performed by functions

Table 14 summarizes the consensus validation scenarios that can lead to block rejection. Achieving full coverage across these cases through fuzzing serves as a strong indicator of both harness quality and input generation effectiveness. As shown in the `Cov` column, most checks are covered, except for those executed later in the `ConnectBlock` logic, including:

- `bad-txns-accumulated-fee-outofrange` — accumulated transaction fees in the block exceed valid limits
- `bad-txns-nonfinal` — block contains a non-BIP68-final transaction
- `bad-blk-sigops` — block exceeds the allowed number of signature operations
- `bad-cb-amount` — coinbase transaction pays more than permitted

Currently, `ConnectBlock` is reached indirectly through the `utxo_snapshot`, `utxo_total_supply`, `tx_pool`, and `tx_pool_standard` harnesses. However, these do not fully exercise the complete range of consensus checks. Developing a dedicated harness specifically for

`ConnectBlock` would enable broader coverage, particularly of the error-handling paths that may expose deeper code behaviors.

```
bool Chainstate::ConnectBlock(const CBlock& block, BlockValidationState& state,
                              CBlockIndex* pindex,
                              CCoinsViewCache& view, bool fJustCheck);
```

The function has the advantage of not applying side effects to the chain state. In particular, the `fjustCheck` parameter allows the caller to validate a block without modifying the UTXO set. Even when `fjustCheck` is false, the chain state can be reused between iterations without degrading performance, since `ConnectBlock` itself does not permanently advance the chain tip in this harness context.

8.3.1. Initialization Function

The initialization routine runs once at the start of the fuzzing campaign and constructs an environment that mimics a running node with a valid chain state. The base chain used for this setup is `regtest`. Below are the key steps performed during initialization:

1. **Enable in-memory filesystem**: The harness activates the in-memory file abstraction (`fsbridge::setEnableMemFS(true)`) so that subsequent file operations occur in memory and are fast and reproducible.
2. **Pre-mine blocks**: As with several existing harnesses, the routine pre-mines a large number of blocks (more than 200) so that coinbase outputs become spendable (coinbase maturity enforces a 100-block delay). Because `ConnectBlock` in this harness does not advance the chain tip, UTXOs consumed by test inputs are drawn from this premined pool.
3. **Create a diverse set of spendable outputs**: To provide the fuzzer with a range of realistic inputs (not only coinbase outputs), the initialization crafts transactions that spend available UTXOs and mines blocks containing those transactions. This yields UTXOs of different ages and types (mature, immature, already spent, etc.), enabling the harness to exercise `ConnectBlock` with valid, invalid, immature, or already spent inputs (double-spending).
4. **Generate varied script types**: The setup produces outputs with different public-key script types. These scripts are selected so that the fuzzer can locate solving scripts and witnesses independently of how the outputs are later spent.
5. **Build an input index for fuzzing**: During initialization the harness records a list of candidate transaction inputs that reference existing UTXOs along with their corresponding unlocking script templates. This index is used by the input-consumption step so the harness can choose valid inputs deterministically during fuzz iterations.

8.3.2. Input Format

The fuzz input consists of a single block consumed by the `FuzzedDataProvider`. We provide a custom `ConsumeBlock` routine that reads every header field from the input and also consumes the block's transactions. The mutator is given broad freedom to modify block contents, however, when certain booleans are set, specific fields are overridden with valid values (for example

`hashMerkleRoot`, `hashPrevBlock`, or `nNonce`) to produce well-formed blocks. Aside from the coinbase, a consumed block may contain up to five additional transactions. Each transaction is created by `ConsumeTransaction`, which, like `ConsumeBlock`, lets the mutator alter every transaction field. To improve the likelihood of generating meaningful, valid inputs, transaction inputs are selected from the UTXO list produced during initialization, individual fields of each input can still be modified or left intact according to the fuzzer's choices. For each `COut`, the mutator may choose a script type such as `P2WSH`, `P2SH`, `TAPSCRIPT`, or no script at all. The `ConsumeTransaction` implementation is provided in Appendix G..

Finally, using boolean flags exposed by the data provider, header fields can be adjusted to remain consistent with the transactions contained in the block.

8.3.3. Harness Logic

The core logic of the harness is partially shown below:

```
FUZZ_TARGET(connect_block, .init = initialize_connect_block)
{
    // Initialize data provider
    FuzzedDataProvider fuzzed_data_provider(buffer.data(), buffer.size());

    // [...] setup chain state

    CBlock block = ConsumeBlock(fuzzed_data_provider, *listBlocks.back(), active_tip->nHeight + 1,
    additionnalUTXO);
    CBlockHeader curr_header = block.GetBlockHeader();

    // Fast block rejection
    BlockValidationState state;
    const auto& consensus = g_setup->m_node.chainman->GetConsensus();
    if (!CheckBlock(block, state, consensus)) {
        DEBUGOUTPUT(std::cout << "Block invalid: " << state.GetRejectReason() << std::endl);
        return;
    }

    // [...]

    bool success = active_chainstate.ConnectBlock(block, state, &new_index, active_coins, /*
    justCheck*/ false);

    if (success) {
        Assert(active_chainstate.DisconnectBlock(block, &new_index, active_coins) == DISCONNECT_OK);
    }
}
```

The block is first consumed by the data provider and passed to `CheckBlock` for an initial round of validation. If this check fails, the harness exits early. Otherwise, the validated block is submitted to `ConnectBlock`. Since `ConnectBlock` returns a boolean value, the harness subsequently calls `DisconnectBlock` whenever the block is successfully accepted.

8.3.4. Harness Results

The results were obtained *via* ensemble fuzzing using AFL++, Honggfuzz, and libFuzzer over a 24-hour campaign. Table 15 summarizes the coverage achieved with LLVM Coverage, compared against the baseline coverage provided by the `qa-asset` corpora.

	Functions	Instances	Regions	Lines	Branches	MCDC
Base	6155	12476	54995	75792	26815	2811
New	6168	12558	55222	76095	26960	2820
Diff	**+13** *(+0.15%)*	**+82** *(+0.33%)*	**+227** *(+0.24%)*	**+303** *(+0.26%)*	**+145** *(+0.32%)*	**+9** *(+0.16%)*

Table 15: LLVM Coverage improvement for `connect_block`

The results are encouraging, as several new functions and template instantiations have newly been covered. However, many of these pertain to the harness itself, particularly the new virtual filesystem class introduced above. Table 16 lists the newly covered functions.

Function	#NewCov	#Regs	#Br
DeploymentActiveAfter(CBlockIndex const, Consensus::Params const&, Consensus::BuriedDeployment, VersionBitsCache&)*	No	+2	+1
CTxMemPool& inline_assertion_check<true, CTxMemPool*&>(CTxMemPool*&, char const*, int, char const*, char const*)*	Yes	+8	+0
utxo_snapshot.cpp:fs::exists(fs::path const&)	No	+2	+1
AllocateFileRange(_IO_FILE, unsigned int, unsigned int)*	No	+10	+1
node::BlockManager::ReadBlockUndo(CBlockUndo&, CBlockIndex const&)	Yes	+9	+0
blockstorage.cpp:node::BlockManager::ReadBlockUndo(CBlockUndo&, CBlockIndex const&)::...	Yes	+13	+0
blockstorage.cpp:fs::exists(fs::path const&)	No	+2	+1
BufferedReader<AutoFile>::BufferedReader(AutoFile&&, unsigned long) requires std::is_rvalue_reference_v<T&&>	Yes	+4	+0
HashVerifier<BufferedReader<AutoFile>>::HashVerifier(BufferedReader<AutoFile>&)	Yes	+2	+0
HashVerifier<BufferedReader<AutoFile>>& HashVerifier<BufferedReader<AutoFile>>::operator>><CBlockUndo&>(CBlockUndo&)	Yes	+1	+0
void Unserialize<HashVerifier<BufferedReader<AutoFile>>, CBlockUndo&>(T&, T0&&)	Yes	+1	+0
void CBlockUndo::Unserialize<HashVerifier<BufferedReader<AutoFile>>> (HashVerifier<BufferedReader<AutoFile>>&)	Yes	+1	+0
void CBlockUndo::Unser<HashVerifier<BufferedReader<AutoFile>>> (HashVerifier<BufferedReader<AutoFile>>&, CBlockUndo&)	Yes	+1	+0
blockstorage.cpp:Wrapper<ScriptCompression, CScript&> Using<ScriptCompression, CScript&>(CScript&)	Yes	+1	+0
blockstorage.cpp:Wrapper<TxOutCompression, CTxOut&> Using<TxOutCompression, CTxOut&>(CTxOut&)	Yes	+1	+0
...	..	..	..
void ScriptCompression::Ser<HashWriter>(HashWriter&, CScript const&)	No	+2	+1
void Serialize<BufferedWriter<AutoFile>, unsigned char>(BufferedWriter<AutoFile>...)	Yes	+1	+0
ApplyTxInUndo(Coin&&, CCoinsViewCache&, COutPoint const&)	Yes	+8	+1
Chainstate::DisconnectBlock(CBlock const&, CBlockIndex const, CCoinsViewCache&)*	Yes	+55	+9
Chainstate::ConnectBlock(CBlock const&, BlockValidationState&, CBlockIndex, CCoinsViewCache&, bool)*	No	+9	+4

Table 16: Newly covered or improved function coverage (unexhaustive list)

Many of the newly covered functions correspond to additional template instantiations, particularly for deserialization routines. Importantly, the harness now reaches `ApplyTxInUndo` and `DisconnectBlock`, which were not covered by any previous harness.

File	Functions	Instances	Regions	Lines	Branches	MCDC
src/compressor.h	5	24 (+3)	19	37	6	0
src/deploymentstatus.h	4	4	23 (+1)	16	6 (+1)	0
src/hash.h	19	60 (+7)	30	87	4	0
src/node/blockstorage.cpp	35 (+1)	35 (+1)	360 (+7)	411 (+15)	125 (+3)	5 (+1)
src/primitives/transaction.h	43	122 (+1)	106	143	55	14
src/serialize.h	118	2614 (+50)	236	455	76	3
src/streams.h	64 (+3)	340 (+4)	206 (+14)	288 (+18)	87 (+5)	5
src/sync.h	25	98 (+1)	51	62	7	0
src/test/fuzz/connect_block.cpp	7 (+7)	7 (+7)	151 (+151)	216 (+216)	92 (+92)	5 (+5)
src/uint256.h	29	68 (+1)	52	67	11	3
src/undo.h	4	18 (+3)	13	20	4	0
src/util/check.h	5	76 (+2)	19	18	5	0
src/validation.cpp	138 (+2)	138 (+2)	3029 (+54)	2764 (+54)	1273 (+38)	126 (+3)
src/validationinterface.cpp	41	54	207	154	29 (+6)	0

Table 17: LLVM Coverage per files for `connect_block`

Table 17 shows coverage updates on a per-file basis. Notably, *validation.cpp*, which contains `ConnectBlock` and `DisconnectBlock`, exhibits significant new coverage. The remaining updates primarily reflect coverage of newly instantiated templates.

fuzzer	inputs	#funcs	#regions	#lines	#inst	#branches	#mcdc
Initial	1/1	1306	6533	9436	1973	2273	65
Honggfuzz	258/2514	234	2008	3568	347	1051	105
AFL++	2/997	0	0	0	0	2	2
libFuzzer	42/9863	30	729	526	53	161	31
Total	**13375**	**1570**	**9270**	**13530**	**2373**	**3487**	**203**

Table 18: Inputs and coverage per Fuzzers on `connect_block`

Then, Table 18 presents the contributions of each fuzzer to the resulting coverage. A dummy input was provided as the seed for the campaign. Surprisingly, AFL++ contributed very little, whereas libFuzzer and Honggfuzz submitted a large number of inputs. The `/` indicates the number of accepted inputs over the total submitted.

In this campaign, libFuzzer submitted 9863 inputs, of which only 42 were accepted and transmitted to the other fuzzers, a very low acceptance ratio of 0.4%. In contrast, Honggfuzz achieved an acceptance rate of around 10%. As explained in Section 7, an accepted input is one that effectively covers new artifacts (regions, branches, MCDC) as measured by LLVM Coverage.

Info

An intermediate harness for `ConnectTip` was also implemented and executed, but it did not provide additional coverage or insights beyond what was achieved with `ActivateBestChainStep`, described below.

8.4. Chain Reorganization #2 (with `ActivateBestChainStep`)

Writing a harness for `ConnectBlock` only covers acceptance checks in the block connection process. However, it does not exercise chain reorganization scenarios, which occur higher in the call chain. As illustrated in Figure 5, the workflow for block acceptance and potential reorganization is as follows:

- `ProcessNewBlock` receives a new block to be validated and added to the tip. The block may arrive directly through P2P or be reconstructed from a compact block along with its associated transactions.
- `AcceptBlock` performs almost all consensus checks by invoking `AcceptBlockHeader` and `CheckBlock`.
- `ActivateBestChain` is then called to switch to the best chain, taking into account the newly received block. It searches for the "most-worked" chain and calls `ActivateBestChainStep` with the new tip. This function is responsible for selecting the new tip to follow.
- `ActivateBestChainStep` performs the actual chain switch by disconnecting blocks from the current tip until the fork point and connecting blocks from the new best chain until reaching the new tip.
- `ConnectTip` and `ConnectBlock` handle adding a new block on top of the current tip.

Under normal conditions, there is only a single "most-worked" chain, so `ActivateBestChainStep` simply connects new blocks on top of the current tip. This harness is specifically designed to test less common scenarios in which two candidate chains exist, requiring a reorganization to switch the tip from one chain to another. By targeting `ActivateBestChainStep`, the harness exercises this reorganization logic. Testing higher in the call chain encompasses more complex logic but also introduces more side effects on the chain state, which must be properly rolled back between iterations.

8.4.1. Harness Design

`ActivateBestChainStep` attempts to set the tip of the current chain state to a target tip provided as a parameter. To do so, it disconnects any blocks that are not part of the target tip chain and then connects any missing blocks. During this process, `ActivateBestChainStep` retrieves block data from the `BlockManager`. For the harness to function correctly, input blocks must be inserted into the `BlockManager` before calling `ActivateBestChainStep`. In this implementation, blocks are directly inserted using `WriteBlock`. The harness reuses the initialization function described previously.

1. **First Branch Creation**: Read up to three chained blocks from the input and insert them into the `BlockManager`. This forms the first branch from the current tip of the chain state.
2. **Switch to First Branch**: Call `ActivateBestChainStep` to switch to the tip of this first branch, as it represents the chain with the most work.
3. **Second Branch Creation**: Read additional chained blocks (up to five) from the same origin as in step 1 and insert them into the `BlockManager`. This forms a second branch diverging from the same common ancestor.

4. **Reorganization**: Call `ActivateBestChainStep` again to switch to the second branch if it has more work than the current tip. If so, blocks from the first branch are disconnected before connecting blocks from the second branch.

This harness is significantly more complex than the previous one because the input must contain two independent chains of valid blocks to trigger a reorganization. It also interacts directly with the `BlockManager`, which normally stores blocks on disk. By using the `MemoryFS` abstraction, all filesystem side effects occur in memory, avoiding disk I/O.

To ensure a clean state for each iteration, the chainstate is reset to the last pre-mined block by disconnecting any additional blocks on top of it.

Warning

Disconnecting blocks at each iteration prevents rebuilding the full chainstate, but introduces slight instability. Blocks added to the `BlockManager` cannot be easily removed, so they remain stored. This should have minimal impact because `ActivateBestChainStep` only accesses blocks in the chain of the current or target tip. One side effect of setting a tip that does not belong to the most-worked chain is that some checks performed by `AcceptBlock` may fail. This is one reason why `AcceptBlock` is not used in this harness to insert blocks into the `BlockManager`. Using `AcceptBlock` would require implementing safe removal of blocks or recreating a clean environment, both of which would severely impact performance.

8.4.2. Input Format

The input of the harness is made of two independent chains of blocks. The are consumed from the input with both `ConsumeBlock` and `ConsumeTransaction` functions described previously. The function `activate_best_chain_step` gather blocks from the block manager. As such, they have to be written into it first. However, since blocks are not removed from the block manager at the end of an iteration, we must enforce a valid `hashMerkleRoot`, `hashPrevBlock` and `nNonce`. If a block with the same hash is already present in the block manager, it is not added twice. Due to this design choice, the exact path taken by an input might be impacted by a previous run[26]. However the outcome is not expected to change.

Also, to increase the possibility to create a chain of genuine transactions, the method `ConsumeTransaction` is slightly changed in order to select transactions inputs either from premined block output (as previously), but also from outputs of previous transactions contained in the chain currently being followed. All outputs of `ConsumeTransaction` are added in a list of transactions that can be reused by transactions of the upcoming blocks.

8.4.3. Harness Results

Coverage results, summarized in Table 19, are encouraging given the prior saturation achieved by existing harnesses. The *Base* column in the table represents the union of coverage from all 221+ existing harnesses. This new harness is able to cover 578 additional previously

[26]namely the block is already present and will not be added in the block manager

uncovered lines of code and more than 32 new functions, demonstrating its effectiveness in exercising parts of the codebase that were previously unreachable.

	Functions	Instances	Regions	Lines	Branches	MCDC
Base	6155	12476	54995	75792	26815	2811
New	6187	12604	55480	76370	27061	2838
Final	**+32** *(+0.36%)*	**+128** *(+0.51%)*	**+485** *(+0.49%)*	**+578** *(+0.49%)*	**+246** *(+0.53%)*	**+27** *(+0.43%)*

Table 19: LLVM Coverage improvement for `activate_best_chain_step`

A partial yet detailed view of the newly covered functions is presented in Table 20. Notably, this harness exercises critical functions such as `CTxMemPool::removeForReorg`, `ApplyTxInUndo`, `DisconnectTip`, `ActivateBestChainStep`, and `MaybeUpdateMempoolForReorg`, the last of which restores rolled-back transactions to the mempool.

Function	#NewCov	#Regs	#Br
CBlockFileInfo::AddBlock(unsigned int, unsigned long)	No		+1
DeploymentActiveAfter(CBlockIndex const, Consensus::Params const&, Consensus::BuriedDeployment, VersionBitsCache&)*	No	+2	+1
CChain::FindFork(CBlockIndex const)*	No	+2	+1
FileCommit(_IO_FILE)*	No	+2	+1
DisconnectedBlockTransactions::LimitMemoryUsage()	Yes	+4	+1
DisconnectedBlockTransactions::DynamicMemoryUsage()	Yes	+1	
DisconnectedBlockTransactions::AddTransactionsFromBlock(...)	Yes	+12	+2
DisconnectedBlockTransactions::removeForBlock(...)	No	+5	+1
DisconnectedBlockTransactions::clear()	Yes	+1	
DisconnectedBlockTransactions::take()	Yes	+1	
disconnected_transactions.cpp:RecursiveDynamicUsage(CTxIn const&)	Yes	+5	+1
disconnected_transactions.cpp:RecursiveDynamicUsage(CScript const&)	Yes	+1	
node::CBlockIndexWorkComparator::operator()(CBlockIndex const, CBlockIndex const*)*	No	+2	+1
node::BlockManager::ReadBlockUndo(CBlockUndo&, CBlockIndex const&)	Yes	+9	
bool transaction_identifier<false>::operator==<uint256>(uint256 const&)	Yes	+1	
transaction_identifier<false>::Compare(uint256 const&)	Yes	+1	
CuckooCache::cache<uint256, SignatureCacheHasher>::contains(uint256 const&, bool)	No	+2	+1
CuckooCache::cache<uint256, SignatureCacheHasher>::allow_erase(unsigned int)	Yes	+1	
TestLockPointValidity(CChain&, LockPoints const&)	Yes	+12	+1
CTxMemPool::UpdateForDescendants(boost::multi_index::detail::hashed_index_iterator<...>)	Yes	+2	
CTxMemPool::UpdateTransactionsFromBlock(std::vector<uint256, std::allocator<uint256>> const&)	Yes	+27	+2
CTxMemPool::removeRecursive(CTransaction const&, MemPoolRemovalReason)	No	+9	+2
CTxMemPool::removeForReorg(CChain&, std::function<bool (boost::multi_index::detail::hashed_index_iterator<...>))	Yes	+28	+3
txmempool.cpp:CTxMemPool::UpdateForDescendants(boost::multi_index::detail::hashed_index_iterator<...>)	Yes	+1	
indirectmap<COutPoint, CTransaction const>::lower_bound(COutPoint const&)*	Yes	+1	
Chainstate::MaybeUpdateMempoolForReorg(DisconnectedBlockTransactions&, bool)	Yes	+29	+6
ApplyTxInUndo(Coin&&, CCoinsViewCache&, COutPoint const&)	Yes	+8	+1
Chainstate::DisconnectBlock(CBlock const&, CBlockIndex const, CCoinsViewCache&)*	Yes	+55	+9
Chainstate::ConnectBlock(CBlock const&, BlockValidationState&, CBlockIndex, CCoinsViewCache&, bool)*	No	+9	+4
Chainstate::DisconnectTip(BlockValidationState&, DisconnectedBlockTransactions)*	Yes	+57	+8
Chainstate::ActivateBestChainStep(BlockValidationState&, CBlockIndex, std::shared_ptr<CBlock const> const&, bool&, ConnectTrace&)*	No	+8	+2
Chainstate::SetBlockFailureFlags(CBlockIndex)*	No	+2	+1
Chainstate::TryAddBlockIndexCandidate(CBlockIndex)*	No	+2	+2
ContextualCheckBlock(CBlock const&, BlockValidationState&, ChainstateManager const&, CBlockIndex const)*	No	+2	+2
validation.cpp:Chainstate::MaybeUpdateMempoolForReorg(DisconnectedBlockTransactions&, bool)::...	Yes	+43	+5
ValidationSignals::BlockDisconnected(std::shared_ptr<CBlock const> const&, CBlockIndex const)*	Yes	+14	
validationinterface.cpp:ValidationSignals::BlockDisconnected(std::shared_ptr<CBlock const> const&, CBlockIndex const)::...*	Yes	+10	
validationinterface.cpp:ValidationSignals::BlockDisconnected(std::shared_ptr<CBlock const> const&, CBlockIndex const)::...*	Yes	+1	
validationinterface.cpp:void ValidationSignalsImpl::Iterate<ValidationSignals::BlockDisconnected(std::shared_ptr<CBlock const> const&, CBlockIndex con...)	Yes	+7	
...	..	..	..

Table 20: Newly covered or improved function coverage `activate_best_chain_step` (non-exhaustive list)

File	Functions	Instances	Regions	Lines	Branches	MCDC
src/chain.cpp	14	14	82 (+1)	105 (+1)	51 (+1)	9 (+2)
src/chain.h	28	30	140	156	45 (+1)	6 (+1)
src/compressor.h	5	24 (+3)	19	37	6	0
src/core_memusage.h	7	35 (+6)	19	36	9	0
src/cuckoocache.h	15	25 (+1)	57	117	32	2
src/deploymentstatus.h	4	4	23 (+1)	16	6 (+1)	0
src/hash.h	19	60 (+7)	30	87	4	0
src/indirectmap.h	13 (+1)	13 (+1)	13 (+1)	13 (+1)	0	0
src/kernel/ disconnected_transactions.cpp	7 (+5)	7 (+5)	34 (+19)	42 (+33)	13 (+9)	0
src/memusage.h	21 (+2)	75 (+9)	32 (+2)	56 (+6)	6	0
src/node/blockstorage.cpp	35 (+1)	35 (+1)	361 (+8)	411 (+15)	127 (+5)	5 (+1)
src/primitives/transaction.h	43	122 (+1)	106	143	55	14
src/serialize.h	118	2624 (+60)	236	455	76	3
src/streams.h	64 (+3)	340 (+4)	206 (+14)	288 (+18)	87 (+5)	5
src/sync.h	25	98 (+1)	51	62	7	0
src/test/ fuzz/connect_block.cpp	7 (+7)	7 (+7)	209 (+209)	261 (+261)	111 (+111)	6 (+6)
src/txmempool.cpp	74 (+5)	74 (+5)	839 (+59)	848 (+54)	331 (+23)	13
src/uint256.h	29	68 (+1)	52	67	11	3
src/undo.h	4	18 (+3)	13	20	4	0
src/util/check.h	5	76 (+2)	19	18	5	0
src/ util/transaction_identifier.h	19 (+1)	45 (+2)	23 (+1)	23 (+1)	1	0
src/validation.cpp	141 (+5)	141 (+5)	3134 (+159)	2887 (+177)	1323 (+88)	140 (+17)
src/validationinterface.cpp	43 (+2)	58 (+4)	218 (+11)	165 (+11)	25 (+2)	0

Table 21: LLVM Coverage per files for `activate_best_chain_step`

Table 21 provides a file-level view of the coverage improvements. Notably, significant updates appear in *src/validation.cpp* and *src/validationinterface.cpp*, which are critical for chain state management and consensus operations.

Individual contributions of fuzzers in terms of inputs and coverage are provided for reference in Appendix H..

Figure 13 illustrates the evolution of line coverage throughout the fuzzing campaign. Notably, only three inputs produced substantial line coverage breakthroughs, occurring after approximately 10 and 15 hours of fuzzing. This likely reflects the time required for the three fuzzing engines to collaboratively craft inputs capable of triggering a proper chain reorganization scenario.

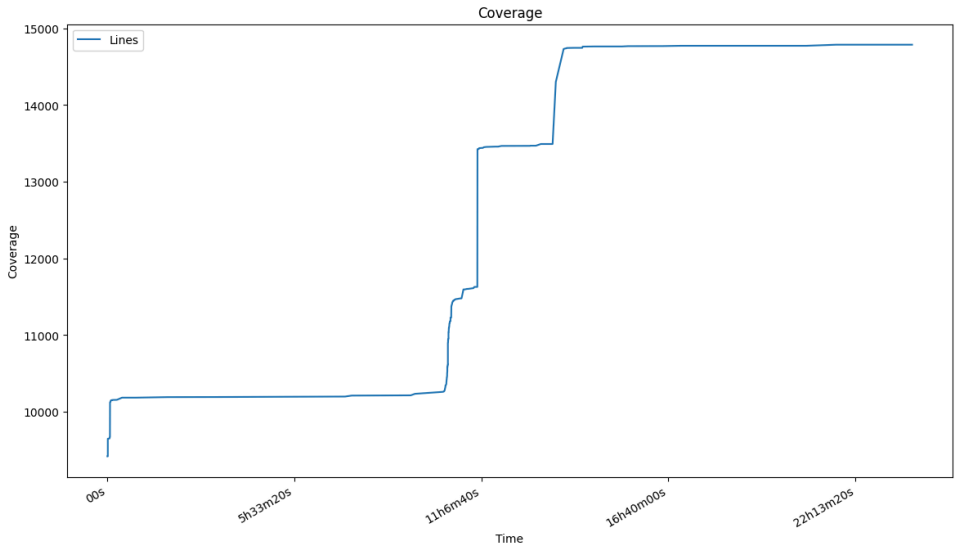


Figure 13: Line coverage evolution for `activate_best_chain_step`

8.5. Chain Reorganization #3 `ActivateBestChain`

After developing the harness for `ActivateBestChainStep`, we targeted `ActivateBestChain`, which is responsible for selecting the chain tip with the most work from all blocks available in the `BlockManager`. By fuzzing this function, we also exercise `FindMostWorkChain`, which implements the logic for identifying the best tip. In this harness, the determination of the longest chain is less influenced by the harness itself and is instead fully guided by the code logic within `FindMostWorkChain`.

8.5.1. Harness Design & Initialization

The design of `ActivateBestChain` is similar to that of `ActivateBestChainStep`. In the harness, we first generate a chain of blocks from the last premined block and add all of them to the `BlockManager` before calling `ActivateBestChain`. We then generate a second chain from the same origin and also insert it into the `BlockManager`. A second call to `ActivateBestChain` may switch the tip to the second chain if it has more work than the current tip. However, it may also revert to the first chain if the second chain is found to be invalid.

Since `FindMostWorkChain` evaluates all blocks present in the `BlockManager`, we cannot retain blocks from previous iterations. This necessitates more extensive clean-up steps, which negatively impacts performance. In this harness, block insertion is handled by `AcceptBlock`, ensuring that blocks are correctly integrated into the chain state for subsequent calls to `FindMostWorkChain`.

Because of this, it is necessary to restore the original chain state whenever a block has been added to the `BlockManager` during the current iteration, maintaining a clean environment for future iterations. Recreating a fresh chain state, however, has a significant performance cost.

To optimize, the harness first consumes all blocks from both chains and validates them upfront using `CheckBlockHeader`, `ContextualCheckBlockHeader`, `CheckBlock`, and `ContextualCheckBlock`. If any block is invalid, the entire input is rejected before modifying the `BlockManager` state. This approach avoids unnecessary chain state reconstruction for inputs containing invalid blocks, greatly improving efficiency.

8.5.2. Experiment with AFL++ Fork-Server

Even with the optimization, the harness remains slow, achieving fewer than 10 executions per second. To address this, we experimented with removing the persistent mode of the harness and instead relying on AFL++'s default fork-server mode. In this mode, the `fork` syscall is used at each iteration to start from a fresh memory state, eliminating the need to restore a clean chain state at the end of each iteration.

Combined with the in-memory filesystem abstraction described earlier, no files need to be erased between iterations. This approach yielded an approximate 2-3 times speed improvement. However, it requires modifying the fuzzing main function and restricts fuzzing to AFL++ only. Given that the performance gain was moderate, this approach was ultimately forsaken.

8.5.3. Harness Input Format

The input format is similar to `ActivateBestChainStep`.

8.5.4. Harness Results

Since this harness is integrated higher in the validation call chain, its resulting coverage is theoretically a superset of `activate_best_chain_step`. Any code exercised by `activate_best_chain_step` should also be exercised by `activate_best_chain`. However, because this harness is more complex and slower, it may achieve less coverage within the same timeframe.

To account for this, we evaluated the incremental improvement provided by this harness over `activate_best_chain_step`. Table 22 summarizes the coverage gains obtained, including the baseline coverage from the 200+ existing fuzzing harnesses. Any newly covered functions, branches, or lines represent code not previously exercised by other harnesses.

	Functions	Instances	Regions	Lines	Branches	MCDC
`activate_best_chain_step`	6187	12604	55480	76370	27061	2838
`activate_best_chain`	6191	12608	55548	76464	27120	2850
Final	**+4** *(+0.04%)*	**+4** *(+0.02%)*	**+68** *(+0.07%)*	**+94** *(+0.08%)*	**+59** *(+0.14%)*	**+12** *(+0.22%)*

Table 22: Coverage improvement between `activate_best_chain_step` and `activate_best_chain`

The coverage impact is entirely positive, with several additional regions and lines exercised. Table 23 provides a detailed view of newly covered and expanded functions. Notably, it includes additional regions and branches within the `CTxMemPool` class.

Function	#NewCov	#Regs	#Br
CBlockFileInfo::AddBlock(unsigned int, unsigned long)	No		
utxo_snapshot.cpp:fs::exists(fs::path const&)	No	+2	+1
DisconnectedBlockTransactions::removeForBlock(...)	No	+2	+1
blockstorage.cpp:fs::exists(fs::path const&)	No	+2	+1
CTxMemPool::UpdateForDescendants(...)	No	+13	+4
CTxMemPool::UpdateTransactionsFromBlock(std::vector<uint256, std::allocator<uint256>> const&)	No	+11	+4
CTxMemPool::UpdateChildrenForRemoval(...)	No	+2	+1
CTxMemPool::UpdateForRemoveFromMempool(...)	No	+6	+3
CTxMemPool::removeConflicts(CTransaction const&)	No	+5	+2
CTxMemPool::removeForBlock(...)	No	+2	+1
CTxMemPool::UpdateParent(...)	No	+7	+3
txmempool.cpp:CTxMemPool::UpdateForRemoveFromMempool(...)	Yes	+1	
Chainstate::MaybeUpdateMempoolForReorg(DisconnectedBlockTransactions&, bool)	No		
Chainstate::FindMostWorkChain()	No	+9	+2
Chainstate::ActivateBestChain(BlockValidationState&, std::shared_ptr<CBlock const>)	No		+2
*ChainstateManager::AcceptBlockHeader(CBlockHeader const&, BlockValidationState&, CBlockIndex**, bool)*	No	+3	+1
validation.cpp:IsCurrentForFeeEstimation(Chainstate&)	No	+2	+1
validation.cpp:Chainstate::MaybeUpdateMempoolForReorg(DisconnectedBlockTransactions&, bool)::...	No	+2	+1
...	..	..	..

Table 23: Function covered improvement for `activate_best_chain` *(unexhaustive list)*

File	Functions	Instances	Regions	Lines	Branches	MCDC
src/chain.h	28	30	140	156	44 (-1)	5 (-1)
src/kernel/ disconnected_transactions.cpp	7	7	35 (+1)	47 (+5)	14 (+1)	0
src/test/fuzz/connect_block.cpp	10 (+3)	10 (+3)	233 (+24)	291 (+30)	123 (+12)	12 (+6)
src/txmempool.cpp	75 (+1)	75 (+1)	870 (+31)	896 (+48)	357 (+26)	16 (+3)
src/validation.cpp	141	141	3146 (+12)	2898 (+11)	1339 (+16)	144 (+4)
src/validationinterface.cpp	43	58	218	165	30 (+5)	0

Table 24: File coverage improvement for `activate_best_chain`

Table 24 shows the coverage improvements per file. While updates in *connect_block.cpp* largely reflect harness-specific code, the gains in *txmempool.cpp* and *validation.cpp* are more significant, as these files play a crucial role in Bitcoin's protocol logic. For instance, Figure 14 illustrates the coverage increase in `removeForBlock` when disconnecting a block, ensuring that all transactions contained within are properly removed from the mempool.

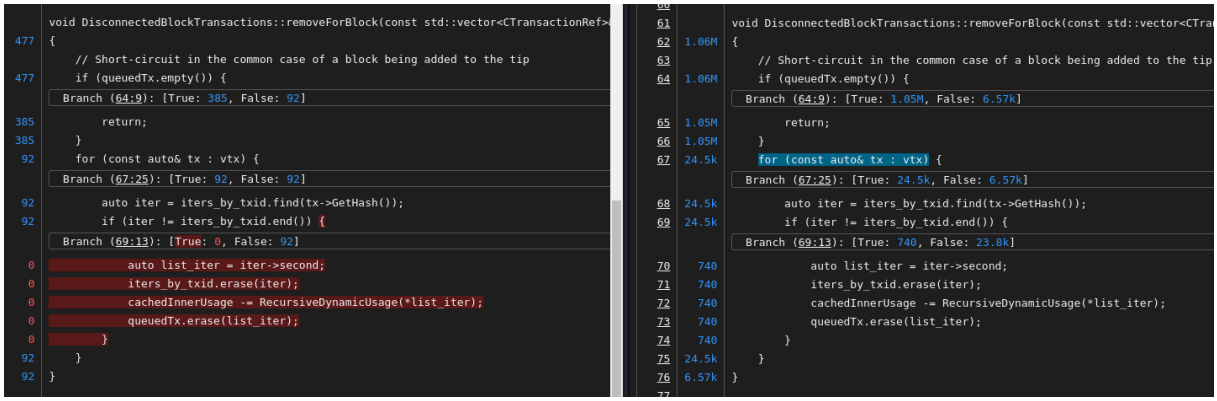


Figure 14: Line coverage improvement for `removeForBlock`

8.6. Overall Results

The three harnesses together contributed to the coverage gains summarized in Table 25. Given the already high coverage saturation, the improvements are notable, with 801 additional lines covered. It should be noted that some of these newly covered lines correspond to the harness code itself, including features such as the virtual filesystem, therefore, the net increase in coverage for the original Bitcoin Core code is slightly lower.

	Functions	Instances	Regions	Lines	Branches	MCDC
Base	6155	12476	54995	75792	26815	2811
New	6194	12611	55655	76593	27164	2858
Final	**+39** *(+0.43%)*	**+135** *(+0.54%)*	**+660** *(+0.68%)*	**+801** *(+0.68%)*	**+349** *(+0.76%)*	**+47** *(+0.79%)*

Table 25: Overall Coverage Improvement new harnesses

The coverage of newly covered or improved functions is detailed in the previous tables. Table 26 provides a consolidated view of all files whose coverage was enhanced by the newly introduced harnesses.

File	Functions	Instances	Regions	Lines	Branches	MCDC
src/chain.cpp	14	14	82 (+1)	105 (+1)	51 (+1)	9 (+2)
src/chain.h	28	30	140	156	45 (+1)	6 (+1)
src/compressor.h	5	24 (+3)	19	37	6	0
src/core_memusage.h	7	35 (+6)	19	36	9	0
src/cuckoocache.h	15	25 (+1)	57	117	32	2
src/deploymentstatus.h	4	4	23 (+1)	16	6 (+1)	0
src/hash.h	19	60 (+7)	30	87	4	0
src/indirectmap.h	13 (+1)	13 (+1)	13 (+1)	13 (+1)	0	0
src/kernel/disconnected_transactions.cpp	7 (+5)	7 (+5)	35 (+20)	47 (+38)	14 (+10)	0
src/memusage.h	21 (+2)	75 (+9)	32 (+2)	56 (+6)	6	0
src/node/blockstorage.cpp	35 (+1)	35 (+1)	361 (+8)	411 (+15)	127 (+5)	5 (+1)
src/primitives/transaction.h	43	122 (+1)	106	143	55	14
src/serialize.h	118	2624 (+60)	236	455	76	3
src/streams.h	64 (+3)	340 (+4)	206 (+14)	288 (+18)	87 (+5)	5
src/sync.h	25	98 (+1)	51	62	7	0
src/test/fuzz/connect_block.cpp	13 (+13)	13 (+13)	340 (+340)	420 (+420)	165 (+165)	18 (+18)
src/txmempool.cpp	75 (+6)	75 (+6)	870 (+90)	896 (+102)	357 (+49)	16 (+3)
src/uint256.h	29	68 (+1)	52	67	11	3
src/undo.h	4	18 (+3)	13	20	4	0
src/util/check.h	5	76 (+2)	19	18	5	0
src/util/transaction_identifier.h	19 (+1)	45 (+2)	23 (+1)	23 (+1)	1	0
src/validation.cpp	141 (+5)	141 (+5)	3146 (+171)	2898 (+188)	1340 (+105)	145 (+22)
src/validationinterface.cpp	43 (+2)	58 (+4)	218 (+11)	165 (+11)	30 (+7)	0

Table 26: Overall File coverage improvement for all harnesses

The most significant improvement from these new harnesses is their ability to generate chain reorganizations, thereby exercising all associated code logic. This covers critical functions such as `DisconnectTip`, `MaybeUpdateMempoolForReorg`, and `ApplyTxInUndo`. Analysis of the resulting coverage highlights the following points:

- Figure 15 illustrates coverage in `UpdateForRemoveFromMempool`, which is invoked when a transaction and its descendants are removed from the mempool. This logic had not been exercised previously.
- Figure 16 shows a branch in `ConnectBlock` related to coinbase amount validation. It ensures that a miner attempting to credit itself with excessive BTC is correctly rejected, a scenario now triggered by the new harnesses.
- Figure 17, also in `ConnectBlock`, demonstrates coverage of consensus-related behavior where transaction inputs are invalidated by `CheckInputScripts`, leading to block rejection.

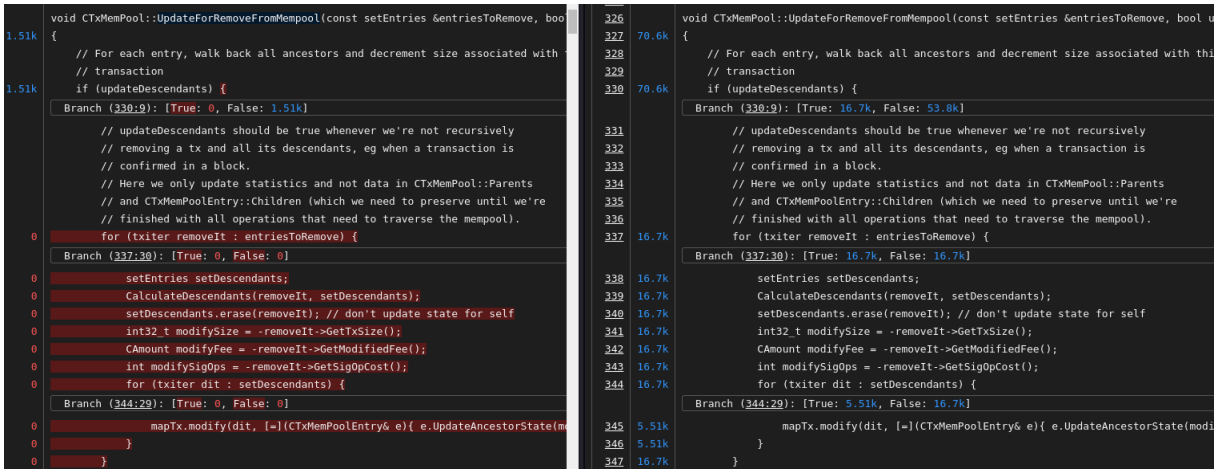


Figure 15: UpdateForRemoveFromMempool Coverage

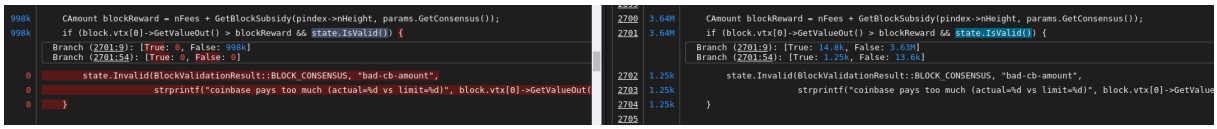


Figure 16: Error bad-cb-amount as a miner tries crediting itself too much BTC

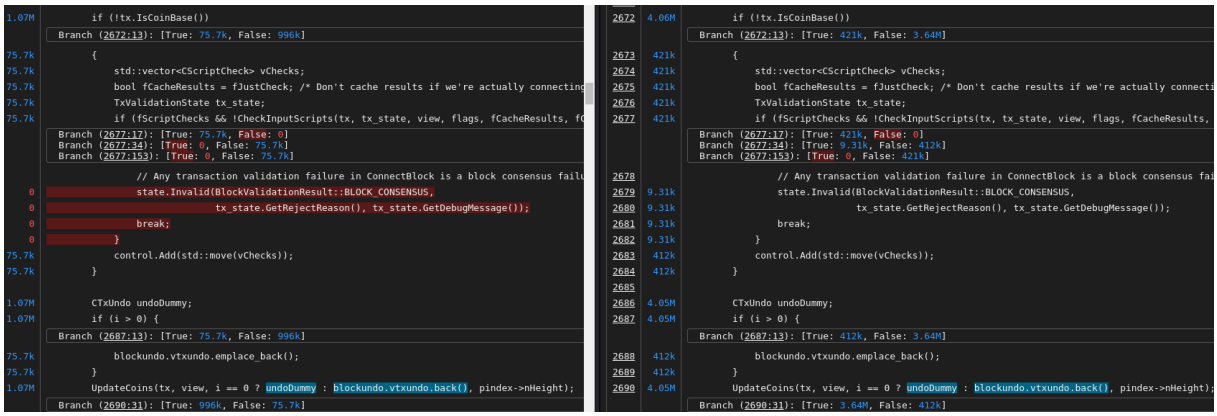


Figure 17: Branch covered in src/validation.cpp *related to* `CheckInputScripts`

9. Structured Fuzzing

Existing harnesses rely on the `FuzzedDataProvider` class to parse the fuzzer input. This module allows consuming arbitrary data types such as booleans, integers, or byte arrays. While it provides a convenient abstraction for implicit structuring of inputs, it also introduces several limitations:

- For a given input, it's difficult to know which bytes will be used and where. That is particularly true for harnesses like `process_message` where bytes can take a whole different meaning depending on the message type. One cannot obtain information reading the input bytes and have to resort in using a logging function in the harness.
- It is complex to manually craft inputs, so fuzzing needs to start fuzzing from an empty corpus.
- `FuzzDataProvider` does not consume bytes in a linear manner. Bytes are read from the beginning to the end, but booleans are read from the end to the beginning of the input.
- If the harness is changed in any way that changes the input format, it's difficult to create a script to convert the corpus for the new version of the harness.
- When the mutator mutates the input, it might change, add or remove data at any position of the input which will drastically change harness behavior. For example, if `FuzzDataProvider` is used in a loop with `CallOneOf`, an integer will be used to select a function to call in the harness. If all the possible functions do not consume the same amount of data from `FuzzDataProvider`, a simple change of the value used by `CallOneOf` may completely change how the remaining data of `FuzzDataProvider` will be interpreted. As such, it might hinder the reasoning process of the mutator.

With structured fuzzing, we aim to provide a solution to the previous limitations:

- With well-defined format, an external program can easily review the corpus, create new inputs or adapt them for a new version of the harness. When creating a new harness, seed inputs can be generated to cover some known common and corner cases.
- It removes any structurally invalid inputs and thus, might speed up program exploration[27].
- Mutations respect input structure. Duplicating, modifying fields, or adding new entries will be performed in accordance with the defined structure.

The expected outcome is to cover the program more efficiently and if possible covering new paths and behaviors that could lead to new bugs.

9.1. Libprotobuf-mutator Harnesses

To experiment with structured fuzzing, we choose to used Libprotobuf-mutator [21]. Protobuf is a universal format description language. Any protobuf format specification can then be compiled into a fully functional parser and serializer in multiple languages (e.g.: C++, Python,

[27]This assumption is correct if the mutator is having difficulties figuring out the purpose of each byte of the input. But verifying it is particularly difficult and left out of the scope for the evaluation.

Go, Java). The language allows defining structures called messages that can contain basic types (e.g.: integers, booleans, bytes), as well as enums, nested messages or repeated fields (arrays).

Libprotobuf-mutator provides a library for mutating protobuf messages while preserving their structural validity. It offers a C++ API, making it straightforward to integrate into the existing fuzzing workflow.

Info

A previous attempt to integrate Libprotobuf-mutator into Bitcoin Core had already been made [19]. However, it could not be reused, as both Libprotobuf-mutator and Bitcoin Core have since evolved significantly, leading to major incompatibilities.

AFL++ supports custom mutators as external shared libraries, whereas libFuzzer does not. To ensure compatibility with both fuzzers, Libprotobuf-mutator was integrated directly into the fuzzing target rather than as a separate module.

To experiment with Libprotobuf-mutator, we created a new binary named `fuzzproto`, analogous to the existing fuzz binary used for standard fuzzing. To allow fuzzproto to run seamlessly with libFuzzer, we defined a custom macro that merges the functionality of Bitcoin Core's `FUZZ_TARGET` macro with Libprotobuf-mutator's `DEFINE_PROTO_FUZZER`. For AFL++, the workflow differs slightly. The latter requires the custom mutator to be provided as a shared library (compiled without sanitizers). Consequently, to perform structured fuzzing with AFL++, the project must be compiled twice - once to build the custom mutator library (without sanitizers), and a second time to build the `fuzzproto` target binary.

Both the `fuzzproto` binary and the custom mutator library read the environment variable `FUZZ=<harness_name>` to determine which harness to execute and mutate. Due to compilation and linkage issues with Libprotobuf-mutator, a dedicated Docker image was created to reliably build the structured fuzzing targets for both libFuzzer and AFL++.

Each structured harness consists of two components - a `.proto` file defining the protobuf message schema, and a corresponding `.cpp` file implementing the fuzzing logic that consumes the parsed message. Using this setup, structured versions of the following fuzzing harnesses have been implemented:

- `addrman`
- `addrman_serdeser`
- `coins_view`
- `coinscache_sim`
- `data_stream_addr_man`
- `ephemeral_package_eval`
- `eval_script`
- `process_message`
- `process_messages`
- `script_flags`
- `script_interpreter`

- `tx_package_eval`
- `tx_pool`
- `tx_pool_standard`
- `txdownloadman`
- `txdownloadman_impl`
- `txorphan`
- `utxo_total_supply`

These harnesses were selected based on discussions with Bitcoin Core developers and judged to be the most relevant for harnesses. Each structured harness is functionally equivalent to its mutation-based counterpart. The example below shows the protobuf message definition for a transaction, which is reused by several harnesses.

```
message CComplexeTransaction {
  message TxInput {
    required uint32 indexTx = 1;
    required uint32 indexTxOut = 2;
    required Sequence sequence = 3;
    message UnlockScript {
      required CScript script = 1;
      repeated bytes scriptWitness = 2;
    }
    optional UnlockScript unlockScript = 4;
  }
  message TxOutput {
    required uint64 amount = 1;
    optional CScript script = 2;
  }
  optional uint32 version = 1;
  required uint32 nLockTime = 2;
  repeated TxInput inputs = 3;
  repeated TxOutput outputs = 4;
}
```

The underlying data structures are modeled recursively as protobuf messages. To emulate `CallOneOf`, which selects an action based on an integer from the input, we use protobuf's `oneof` construct. This lets the fuzzer produce a single, well-formed variant that maps directly to an action. For example, the `addrman` harness can express its selectable actions like this:

```
message Target_addrman {
  // [snip]

  message Action {
    message EmptyAction {}
    message AddAddressesAction {
      repeated proto_fuzz_util_net.Address addrs = 1;
      required proto_fuzz_util_net.NetAddress source = 2;
      required uint32 timePenality = 3;
    }
    message GoodAction {
```

```
      required proto_fuzz_util_net.Service serviceAddr = 1;
      required uint32 time = 2;
    }
    message AttemptAction {
      required proto_fuzz_util_net.Service serviceAddr = 1;
      required bool count_failure = 2;
      required uint32 time = 3;
    }
    message ConnectedAction {
      required proto_fuzz_util_net.Service serviceAddr = 1;
      required uint32 time = 2;
    }
    message SetServicesAction {
      required proto_fuzz_util_net.Service serviceAddr = 1;
      required proto_fuzz_util.WeakEnum n_service = 2;
    }

    oneof action_type {
      EmptyAction ResolvConflict = 1;
      EmptyAction SelectTriedCollision = 2;
      AddAddressesAction AddAddresses = 3;
      GoodAction GoodAddress = 4;
      AttemptAction AttemptAddress = 5;
      ConnectedAction ConnectedAddress = 6;
      SetServicesAction SetServices = 7;
    }
  }

  repeated Action actions = 6;
  // [snip]
}
```

9.2. Results and Comparison with existing Harnesses

The harnesses were executed with libFuzzer and AFL++ within PASTIS. Individual campaign results are given in Table 27.

Harness	Corpus	Functions	Instances	Regions	Lines	Branches	MCDC
activate_best_chain	3883	1519	8783	11971	2394	3471	216
addrman	4155	684	3126	4542	951	1274	102
addrman_serdeser	3361	562	2569	3745	773	1036	85
coins_view	6158	512	2043	2893	643	915	79
coinscache_sim	2175	214	969	1496	225	455	22
ephemeral_package_eval	15828	977	4988	6295	1420	1769	77
eval_script	7475	298	1600	2471	307	961	116
process_message	3532	1195	6774	8337	2080	2460	191
process_messages	17507	1428	8483	10265	2501	3208	278
script_flags	8081	539	3464	5215	639	1390	151
script_interpreter	1412	356	1083	1873	448	392	28
tx_package_eval	16157	1121	5632	7326	1624	2196	124
tx_pool	13548	1454	8289	11454	2004	3274	263
tx_pool_standard	15866	1151	5740	7561	1683	2060	84
txdownloadman	5565	568	2061	3268	727	804	54
txdownloadman_impl	5067	560	2203	3331	719	845	56
txorphan	11424	301	1106	1710	404	356	7
utxo_total_supply	5057	1354	7723	10503	2408	2503	104

Table 27: Coverage obtained by Structured-based Harnesses

Manual comparison with existing harnesses indicates that the structured harnesses generally underperform in terms of coverage. This is largely due to the extensive fuzzing hours already accumulated by the existing harnesses, whereas the structured harnesses started from an empty corpus and were run for approximately 24 hours each.

A more meaningful comparison can be made with the `activate_best_chain` harness, which was also created from scratch and started from an empty corpus. Table 28 summarizes the coverage differences. While structured fuzzing with Libprotobuf-mutator does show some coverage improvements, it also introduces a significant number of additional functions into the codebase.

	Functions	Instances	Regions	Lines	Branches	MCDC
FuzzDataProvider	1387	2268	8536	11413	3490	260
Structured	1519	2394	8783	11971	3471	216
Final	**+132** *(-5.52%)*	**+126** *(+0.92%)*	**+247** *(+1.05%)*	**+558** *(-0.91%)*	**-19** *(+1.03%)*	**-44** *(+0.59%)*

Table 28: Coverage improvement between FuzzDataProvider and Libprotobuf-mutator on `activate_best_chain`

Manual inspection of the intersection of covered functions shows more nuanced results. For example, the mutational approach achieved broader coverage in `VerifyWitnessProgram` as

shown in Figure 18, whereas the structured harness captured certain malleability scenarios in `CheckWitnessMalleation` as shown in Figure 19.

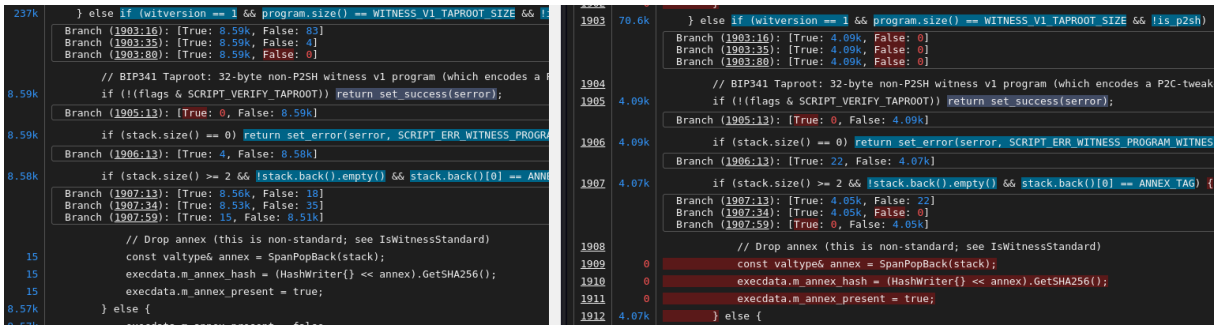


Figure 18: VerifyWitnessProgram coverage

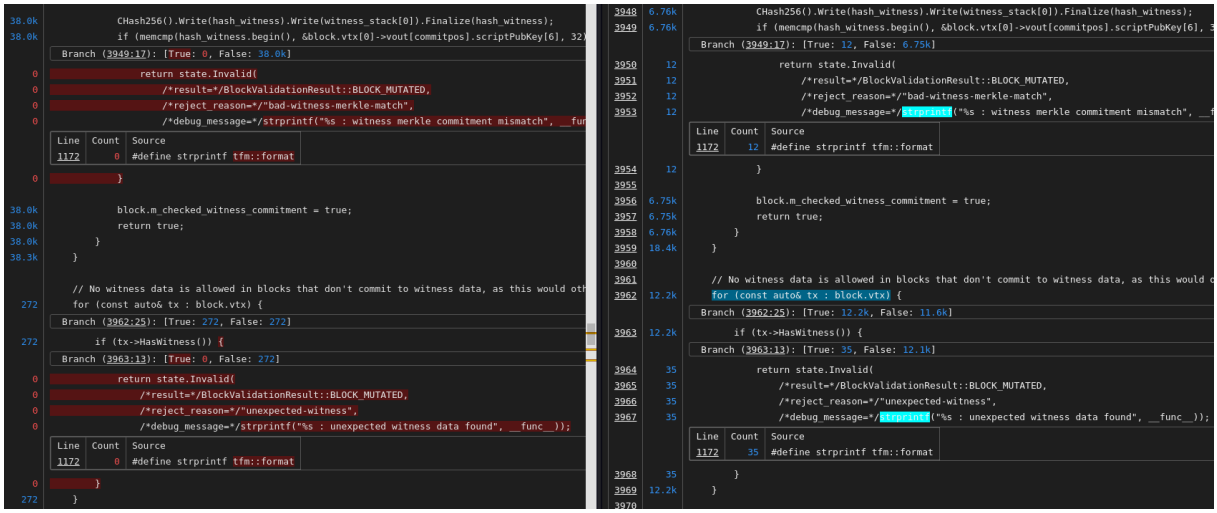


Figure 19: CheckWitnessMalleation coverage

Ultimately, considering the intersection of covered functions, the FuzzDataProvider approach covered **165** additional regions and **58** more branches for `activate_best_chain`. A similar experiment on the `addrman` harness, starting from an empty corpus for the same duration, again showed that the purely mutational approach outperformed the protobuf-based harness while maintaining identical input semantics. In this scenario, the most likely explanation is the computational overhead introduced by protobuf deserialization compared to `FuzzDataProvider`, which operates at a lower level and is faster.

A thorough investigation of potential performance bottlenecks in libprotobuf-mutator is required before drawing definitive conclusions on this approach. This has been deferred as future work due to time constraints.

9.3. Miniscript Grammar-based Fuzzing

Miniscript [44] is a high-level language for expressing Bitcoin scripts and policies in a readable, composable form. Miniscript programs can be compiled into Bitcoin Script bytecode that

the Bitcoin Script interpreter executes. It provides a comprehensive way to express spending conditions (e.g., multisignature thresholds, timelocks, and combinations thereof).

Bitcoin Core developers have noted that fuzzers struggle to cover deep and complex Bitcoin Script interpreter logic. While miniscript is very well covered it might be used to generate complex and relevant scripts to be used in the interpreter that would for instance exercise complex multi-signature or timelock spending conditions. For this, reason we explored grammar-based fuzzing as a mean to generate relevant scripts.

The `miniscript` harness is a natural candidate for grammar-based fuzzing because its input is a textual language The harness currently tests the compilation pipeline down to the generated opcode bytes. Integrating grammar-based mutation into transaction-producing harnesses could theoretically accelerate exploration by producing many more relevant scripts if reused to exercise deeper interpreter logic.

For this purpose we used Grammar-Mutator [45]. The workflow is:

1. Define a JSON grammar describing the language surface (terminals, nonterminals, and production rules).
2. Compile the JSON grammar into an ANTLR grammar.
3. Use the generated grammar to build an external mutator as a shared library that plugs into AFL++ as a drop-in replacement for its default mutator.

This method requires no changes to the harness itself, the harness continues to accept textual Miniscript inputs, while the external mutator produces syntactically and structurally valid variants.

Miniscript's language distinguishes typed expressions. Valid expression types include *Base*, *Verify*, *Key*, and *Wrapped*. For example, `pk_k(K)` accepts only *key* expressions as its argument, whereas `hash256(B)` accepts a *base* expression. The language includes operators and constructors such as `sha256`, `andor`, `or_c`, and `multi`.

The example below encodes a 3-of-3 threshold spending policy that becomes a 2-of-3 policy after a 90-day timelock.

```
thresh(3,pk(key_1),s:pk(key_2),s:pk(key_3),sln:older(12960))
```

Once compiled the resulting bytecode is the following:

```
<key_1> OP_CHECKSIG OP_SWAP <key_2> OP_CHECKSIG OP_ADD OP_SWAP <key_3>
OP_CHECKSIG OP_ADD OP_SWAP OP_IF
  0
OP_ELSE
  <a032> OP_CHECKSEQUENCEVERIFY OP_0NOTEQUAL
OP_ENDIF
OP_ADD 3 OP_EQUAL
```

The example below gives an extract of the grammar used to define the language for the threshold function:

```
{
    "<onenine>": [["1"], ["2"], ["3"], ["4"], ["5"], ["6"], ["7"], ["8"], ["9"]],
    "<digit>": [["0"], ["<onenine>"]],
    "<digit-1>": [["<digit>"], ["<digit>", "<digit-1>"]],
    "<digits>": [["<digit-1>"]],

    "<thresh-1>": [[",", "<thresh-1-1>"]],
    "<thresh-x>": [[], ["<thresh-1>"]],
    "<thresh-1-1>": [["<WrapExpr>", "<thresh-x>"]],
    "<thresh>": [["thresh(", "<digits>",",", "<BaseExpr>", "<thresh-1>", ")"]],
    //...
}
```

The grammar follows the given logic:

- elements within a list are interpreted sequentially *(and should be present)*
- elements list-of-lists represent alternative options from which one is chosen.

Recursion plays a crucial role in defining complex, nested constructs. However, a key limitation of this approach is the inability to define dependent fields. For example, in the `thresh` function, the first argument `k` specifies how many keys follows. However, the grammar here cannot enforce a specific number of keys when defined as recursive items. Without hardcoding the number of keys, the grammar may still generates invalid scripts that the interpreter will reject. The complete grammar used for the experiments is provided in Appendix E.. Once the external mutator is compiled as a shared library, it can be used with AFL++ as follow:

```
export AFL_CUSTOM_MUTATOR_LIBRARY=./libgrammarmutator-miniscript.so
#export AFL_CUSTOM_MUTATOR_ONLY=1
export FUZZ=miniscript_string

afl-fuzz -M main -i inputs/ -o outputs -c bins/fuzz.afl.cmplog - ./bins/fuzz.afl @@
```

The harness was initially executed with `AFL_CUSTOM_MUTATOR_ONLY=1`, which forces the fuzzer to rely exclusively on the grammar-based mutator. However, subsequent experiments showed that combining the grammar-based mutator with AFL++'s default mutator significantly improved fuzzing efficiency.

9.3.1. Fuzzing Results

Fuzzing was conducted using both the existing `miniscript_string` harness and the grammar-based mutator. To fairly assess their impact, both campaigns were started from an empty corpus.

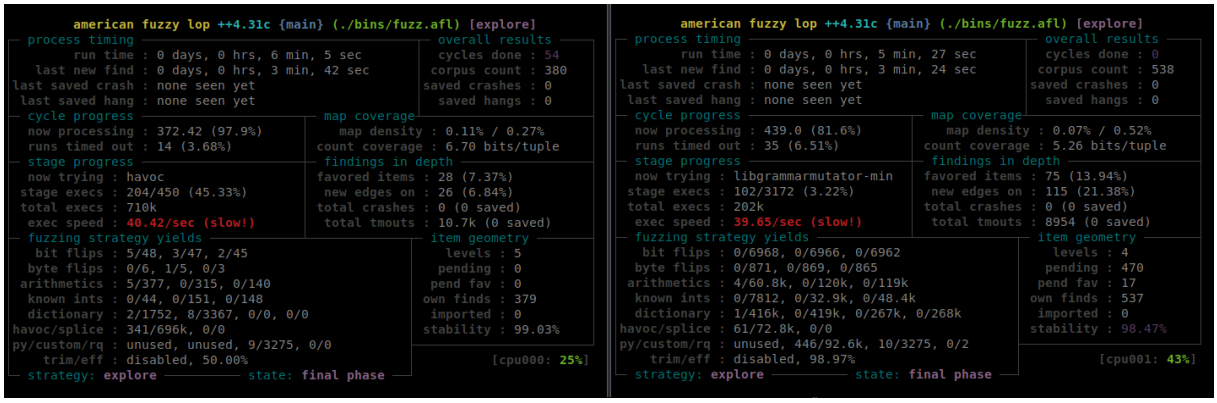


Figure 20: Mutational and Grammar-based Mutations

Figure 20 shows AFL++ running in both modes from an empty corpus. The grammar-based fuzzing achieves higher coverage within just a few minutes. While this approach is more efficient, it proved somewhat unstable, with AFL++ frequently interrupting itself due to a known bug[28].

	Functions	Instances	Regions	Lines	Branches	MCDC
Existing	145	165	1423	1672	1080	100
Grammar	145	165	1412	1669	1036	88
Final	-	-	**-11**	**-3**	**-44**	**-12**

Table 29: Existing Coverage against the one Generated by Grammar Fuzzing

Table 29 compares the coverage of the existing `miniscript_string` harness using the `qa-asset` corpus (634 files) with the corpus generated by grammar-based fuzzing in approximately 10 minutes (548 files). While the grammar-based corpus is slightly smaller, it manages to cover the same number of functions and template instances, as well as 99.82% of the lines covered by the `qa-asset` corpus, in just a few minutes. This indicates that a grammar-based approach for this harness is significantly more time-efficient than a purely mutational approach.

While this approach was not experimented to supersede existing miniscript harnesses, the fuzzing instability and due to the limitations discussed above it was not feasible to further use the generated corpus for interpreter fuzzing.

9.4. Conclusion

While a previous attempt to integrate Libprotobuf-mutator (LPM) in Bitcoin Core has been made [19], several observations can be drawn from these experiments. First, adding LPM as an external dependency can be cumbersome due to version incompatibilities. For this reason, the mutator was fully integrated and compiled within the `fuzz` binary rather than being used externally. The main advantages of LPM are the ability to craft inputs ahead of fuzzing and improved interpretability of generated inputs, which alleviates the current limitation of starting from an empty corpus.

[28] https://github.com/AFLplusplus/Grammar-Mutator/issues/35

Structured fuzzing is expected to expose more complex bugs requiring structured inputs. However, in practice, this approach did not trigger any significant bugs or hangs. Longer fuzzing campaigns would likely be needed to draw more meaningful conclusions. The experiments were run for approximately 24 hours[29], a relatively short time given the size and complexity of the Bitcoin Core codebase, especially for slower harnesses.

Similar conclusions apply to grammar-based fuzzing. While it enables generating readable and semantically valid inputs, it applies only to specific harnesses, introduces additional integration complexity, and restricts the fuzzer to AFL++.

Finally, the potential benefits of structured fuzzing are partially mitigated by the large computational resources already provided by OSS-Fuzz.

[29]on a servers with Intel Xeon E3 (20 cores) and 70Gb RAM

10. Cryptographic Primitives Testing

Cryptographic algorithms in Bitcoin Core are fundamental to the system's overall security. Any flaw or weakness in their implementation could lead to severe vulnerabilities, including unauthorized access to funds, data breaches, or denial-of-service attacks. Rigorous testing is therefore essential to ensure correctness, security, and compliance with established standards.

Implementing cryptographic algorithms correctly is challenging due to constraints that vary from input validation to constant-time execution to prevent side-channel leaks.

- Google's Wycheproof -> Project Wycheproof. As a first step, algorithms must be validated against standard test vectors provided by organizations

like NIST's Cryptographic Algorithm Validation Program [46] or the Wycheproof project [47]. These test suites cover a wide range of scenarios, including edge cases such as zero-length messages and null keys.

The specific testing targets were selected in collaboration with Bitcoin Core developers, who helped prioritize primitives that had not already been extensively tested (for example, `secp256k1`). Consequently, we focused on the project's implementations of the `ChaCha20-Poly1305` authenticated encryption algorithm [48] and the `SHA-256` hashing algorithm [49]. The former secures the second version of the peer-to-peer (P2P) network protocol [50], ensuring message confidentiality and integrity between nodes. The latter underpins multiple components of Bitcoin's proof-of-work and consensus mechanisms, including block chaining and transaction verification. The `SHA-256` implementation also includes architecture-specific variants that are not yet covered by fuzzing.

To test these custom cryptographic implementations, we used crypto-condor [51], an open-source tool developed at Quarkslab that automates validation against standard test vectors. Dedicated harnesses were created to interface directly with the relevant Bitcoin Core algorithms, making them compatible with `crypto-condor` and enabling systematic verification of correctness and robustness.

10.1. Crypto-condor

The Crypto-condor[30] tool is a compliance-testing framework for cryptographic primitives, available both as a Python library and as a command-line interface (CLI). For a given algorithm, crypto-condor executes the relevant test vectors and compares the outputs of the target implementation with the expected results.

For this evaluation of Bitcoin Core's custom cryptographic implementations we used `crypto-condor`'s Harness API. The harness mode operates similarly to libFuzzer - any shared library or Python module that exports functions following the harness API naming convention can be tested directly by `crypto-condor`.

[30] https://quarkslab.github.io/crypto-condor/latest/index.html

For example, to test the `ChaCha20-Poly1305` primitive, the harness library must export an external symbol that resolves to the following function signature:

```
extern "C" int CC_SHA_256_digest(uint8_t *digest, const size_t _,  const uint8_t
*input, const size_t input_size);
```

This function will automatically be recognized by `crypto-condor` and tested. Figure 21 illustrates the architecture of the harness mode for `SHA-256`.

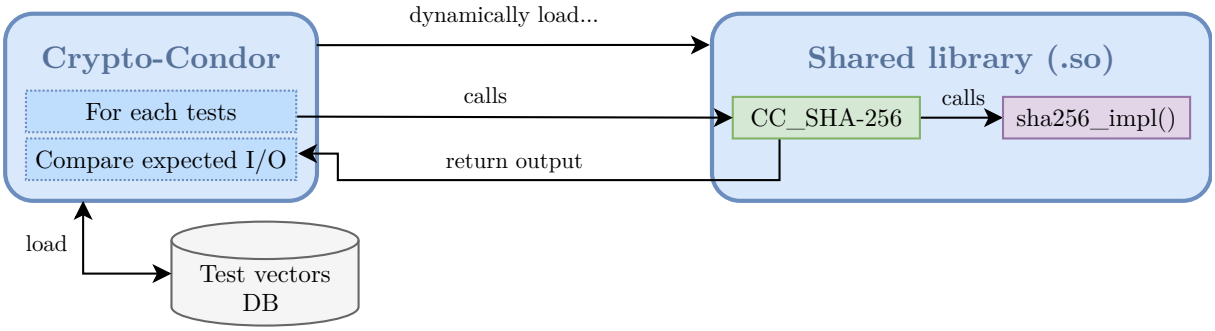


Figure 21: `crypto-condor` harness mode overview

To test the cryptographic implementations, the relevant source files were extracted from the codebase and built into standalone shared libraries, which are dynamically loaded during testing. The following sections describe the two harnesses and present the results obtained using the standard test vectors.

10.2. SHA-256 Harness

SHA-256 is heavily used in Bitcoin. The source code is located in *src/crypto/*. The header *sha256.h* defines the class `CSHA256` that wraps various CPU and architecture specific implementations. Using hardware accelerated instructions is particularly important to achieve optimal performances. Bitcoin supports a pure software implementation, as well as variants using `SHA-NI` and `SSE4.1` instructions on both `x86_64` and `ARM64` (`Aarch64`).

sha256.cpp	96.94% (571/589)	100.00% (19/19)	87.78% (79/90)
sha256_avx2.cpp	100.00% (286/286)	100.00% (24/24)	100.00% (24/24)
sha256_sse4.cpp	100.00% (935/935)	100.00% (1/1)	100.00% (1/1)
sha256_sse41.cpp	100.00% (278/278)	100.00% (24/24)	100.00% (24/24)
sha256_x86_shani.cpp	0.00% (0/286)	0.00% (0/11)	0.00% (0/13)

Figure 22: OSS-Fuzz Coverage SHA256

As illustrated in Figure 22, OSS-Fuzz achieves excellent coverage for SHA-256, with the exception of the `SHA-NI` variant on `x86_64`. The ARM variant, implemented in *sha256_arm_shani.cpp*, is not even listed because it is not compiled into the harness.

To test all five variants combinatorially, we developed a dedicated harness that builds a standalone SHA-256 library. This library exposes three functions, allowing direct invocation of each variant. Normally, the `SHA256AutoDetect` function is used to select the appropriate variant at runtime via `cpuid`, and setting a global function pointer accordingly. In the harness, we

override this mechanism to explicitly select the desired variant. The following function, added in *sha256.cpp*, demonstrates how the global Transform pointer is overridden:

```
extern "C" void do_sha256_x86_shani(unsigned char* out, const unsigned char* in, size_t len)
{
    Transform = sha256_x86_shani::Transform;
    CSHA256 hasher;
    hasher.Write(in, len);
    hasher.Finalize(out);
}
```

Info

The library also implements two additional variants, `SSE4.1` and `AVX2`, which allow computing multiple hashes in parallel (2-way and 4-way) on data aligned to the block size. Because these variants internally handle padding in advance, they do not correspond to standard SHA-256 operations and, consequently, are not exercised by the harness.

The table below summarizes the results for the different CPU architectures and the related implementations using `crypto-condor`:

Host	Software	SHA-NI	SSE4
AArch64	✓	✓	-
x86_64	✓	✓	✓

Table 30: Summary of testing SHA-256 against standard test vectors

The images below illustrate the number of tests performed by `crypto-condor` and the validating output:

Info

Software stands for the standard non-hardware dependent implementation

(a) Soft - `x86_64` *(b) SHA-NI - `x86_64`* *(c) SSE4 - `x86_64`* *(d) Soft - `AArch64`* *(e) SHA-NI - `AArch64`*

Figure 23: Cryptocondor results on SHA-256

Success

All of the implementations passed compliance verification with no discrepancies detected on both `x86_64` and `Aarch64` CPU architectures.

10.3. ChaCha20-Poly1305 Harness

`ChaCha20-Poly1305` is an AEAD (Authenticated Encryption with Associated Data) stream cipher used in BIP324 for end-to-end encryption between nodes. Its implementation in Bitcoin Core is derived from the original version by D.J. Bernstein. Introduced in June 2023, it was chosen for testing as it may have received less scrutiny compared to `secp256k1`.

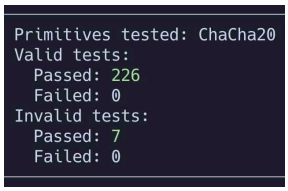
The harness library can simply expose `CC_ChaCha20_Poly1305_encrypt` and `CC_ChaCha20_Poly1305_decrypt`[31], which invoke the underlying functions appropriately. The algorithm is encapsulated in the `AEADChaCha20Poly1305` class, which only needs to be instantiated to call the relevant methods.

The table below summarizes the results for both encryption and decryption operations. While there are no architecture-specific variants, the implementation was tested on both Linux `x86_64` and `AArch64`.

Host	Encrypt	Decrypt
AArch64	✓	✓
x86_64	✓	✓

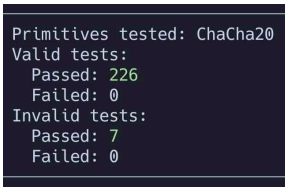
Table 31: Test vectors testing ChaCha20-Poly1305

The images below illustrate the number of tests performed by `crypto-condor` and the validating output:

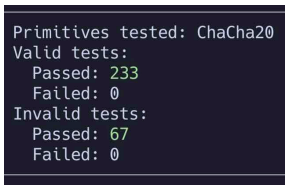


(a) `AArch64` - Encrypt

(b) `AArch64` - Decrypt



(c) `x86_64` - Encrypt



(d) `x86_64` - Decrypt

Figure 24: Cryptocondor results on ChaCha20-Poly1305

Success

Both implementations passed compliance verification with no discrepancies detected on both `x86_64` and `Aarch64` CPU architectures.

10.4. Elligator Swift Encoding Non Distinguishability Harness

As part of BIP324, the Elligator Swift [52] encoding is used to exchange public keys between peers during the key exchange phase. This encoding makes public keys indistinguishable from random strings, helping to prevent traffic profiling and mitigate Bitcoin protocol fingerprinting.

[31] https://quarkslab.github.io/crypto-condor/latest/harness-api/ChaCha20.html#encrypt-with-poly1305

`crypto-condor` can also assess pseudo-random generators by applying a suite of statistical tests implemented in `TestU01` [53]. The harness provided in Appendix D. generates multiple encoded public keys into a file using `secp256k1_ellswift_create`. This file is then processed by `crypto-condor`, which runs the statistical tests to evaluate the quality and randomness of the generated encodings.

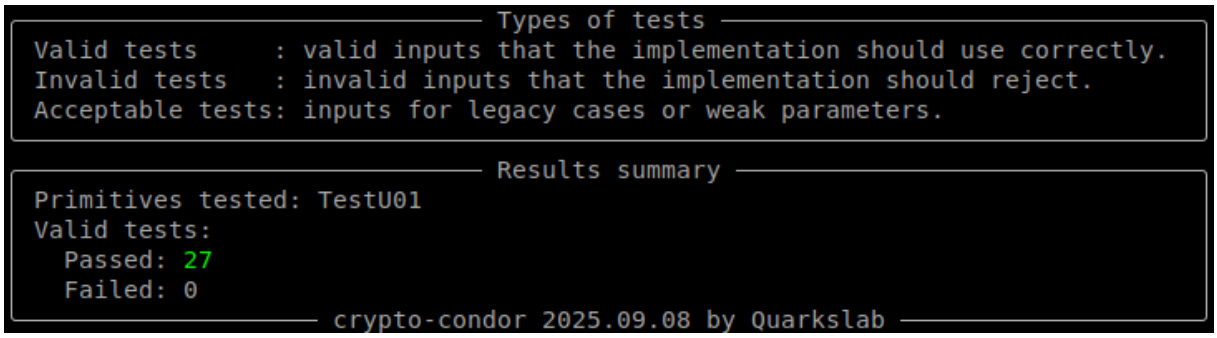


Figure 25: TestU01 on Elligator Swift encoded public keys

As shown in Figure 25, the outputs are considered indistinguishable from random according to all statistical tests performed by `TestU01`.

10.5. Conclusion

Creating standalone libraries and testing them against standard test vectors using `crypto-condor` establishes a foundation for more advanced techniques, such as differential fuzzing. Instead of simply validating an implementation against known input/output pairs, differential fuzzing compares the behavior of two implementations against each other. While no discrepancies were expected given the robustness of the code, this approach provides a strong compliance guarantee and harnesses can be directly reused for differential fuzzing.

Bitcoin Core also includes other cryptographic algorithms, such as MuHash, SHA-512, SHA-3, HMAC, and AES, which were not tested due to time constraints.

11. Differential Fuzzing

Differential fuzzing is a fuzzing technique in which two (or more) independent implementations of the same algorithm are executed with identical inputs, and their outputs are compared. Unlike traditional fuzzing, which primarily seeks crashes or memory safety violations, differential fuzzing aims to identify semantic inconsistencies between implementations. In other words, it detects logical deviations or specification violations even when no crash occurs. This approach goes beyond conventional fuzzing by using one implementation as a reference specification and the other as the target under test. It is especially effective for cryptographic algorithms, which are typically deterministic, mathematically well-defined, and expected to exhibit identical behavior across conforming implementations.

Whereas test-vector validation (see Section 10) relies on a finite set of known-good inputs and outputs, differential fuzzing dynamically generates and mutates inputs to explore broader behavioral space. Progress, as with other traditional fuzzers, is guided by code coverage. In a pure black-box setting, only input/output discrepancies are analyzed, but white-box differential fuzzers can also incorporate coverage feedback to guide input selection more intelligently.

Differential fuzzing can be applied across several dimensions:

- **Different implementations** of the same algorithm (e.g., OpenSSL vs LibreSSL).
- **Different versions** of the same implementation (e.g., to detect regressions).
- **Different configurations** of the same implementation (e.g., with or without hardware acceleration).

In the Bitcoin ecosystem, several initiatives have explored this approach. Cross-implementation differential testing has been used to compare consensus and transaction validation behaviors (see Section 11.4). Cryptographic primitives have also been evaluated through frameworks such as CryptoFuzz [54], while Semsan [18] introduced a differential fuzzer based on LibAFL. Although Semsan is no longer actively maintained, its methodology continues through newer efforts such as Fuzzamoto [20].

For this security assessment, the two previously created harnesses were used for differential fuzzing, comparing Bitcoin Core's implementations against the OpenSSL [55] reference. The differential fuzzing campaigns were executed using DeltAFLy, Quarkslab's in-house differential fuzzing framework.

11.1. DeltAFLy

DeltAFLy [56] is Quarkslab's internal differential fuzzer, developed in Rust and built on top of libAFL [57] for API-level fuzzing. It is specifically designed for testing cryptographic libraries and operations. Inspired by the `cryptofuzz` project [54], DeltAFLy addresses several of its limitations through a more modular architecture where each harness is packaged as an independent shared library.

It supports differential fuzzing across multiple languages, including Rust, C, C++, and Java. Furthermore, thanks to libAFL's QEMU mode, it can also target binary-only implementations. Leveraging libAFL's Observer modularity, DeltAFLy features a constant-time analysis mode capable of detecting secret-dependent control flow. DeltAFLy notably led to the discovery of vulnerabilities in the HQC post-quantum algorithm implementation within LibOQS[32].

The results of the performed tests are presented in the following sections.

11.2. DeltAFLy: ChaCha20-Poly1305 Harness

The `ChaCha20-Poly1305` primitive is implemented in *src/crypto/chacha20poly1305.cpp* within the Bitcoin Core repository. This implementation is not used directly in production, instead, it is encapsulated by the `FSChaCha20Poly1305` wrapper, which adds forward-secrecy properties by incrementing the nonce after each encryption or decryption operation and rotating the symmetric key after a fixed number of uses [58].

For this assessment, we performed a differential comparison of the core `ChaCha20-Poly1305` primitive against OpenSSL's reference implementation. The `FSChaCha20Poly1305` wrapper was not tested in this campaign. However, assessing it in future work would be valuable, as its nonce management and key rotation logic are essential to ensuring forward secrecy and preventing nonce reuse.

To enable interoperability between DeltAFLy and the C++ target, a dedicated driver (also called "module" within the project) was developed to bind the fuzzer to the C++ shared library and invoke the corresponding cryptographic functions with the appropriate parameters. The results returned from these calls were then passed back to DeltAFLy for output comparison.

The C++ harness was built from a minimal extraction of the Bitcoin Core source tree, including only the files required to compile the `ChaCha20-Poly1305` module. Since the target cryptographic code is relatively self-contained, this subset remained small. The resulting shared library was compiled with sanitizer coverage enabled, allowing DeltAFLy to leverage coverage feedback for input selection and guided mutation during fuzzing.

Success

The fuzzing campaign did not uncover any bugs or discrepancies between the two implementations on both x86_64 and Aarch64 CPU architectures

11.3. DeltAFLy: SHA256 Variants (SSE4, SHA-NI)

Using the `SHA-256` harness, we performed differential fuzzing across all five variants of the SHA-256 implementation against OpenSSL's reference version. In addition to the default software-based implementation, we tested the hardware-accelerated variants available for both `AArch64` and `x86_64`, namely `SHA-NI` (for both `x86_64` and `AArch64`) and `SSE4` (`x86_64` only).

[32] https://blog.quarkslab.com/finding-bugs-in-implementations-of-hqc-the-fifth-post-quantum-standard.html

Thanks to the distributed capabilities provided by LibAFL, DeltAFLy can differential fuzz two targets running on separate machines, and therefore on different architectures. In this assessment, however, this feature was not required since the variants were tested in pairs.

As with `ChaCha20-Poly1305`, the harness required implementing a dedicated driver on the DeltAFLy side to interface with the exposed C functions and their various operational modes.

The coverage results obtained using LLVM instrumentation are presented below:

	Baseline			DeltAFLy		
File	**Line**	**Function**	**Region**	**Line**	**Function**	**Region**
sha256.cpp	96.94%	100.00%	87.78%	31.02%	71.43%	32.05%
	(571/589)	(19/19)	(79/90)	(161/519)	(15/21)	(25/78)
sha256_sse4.cpp	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	(935/935)	(1/1)	(1/1)	(935/935)	(1/1)	(1/1)
sha256_x86_shani.cpp	0.00%	0.00%	0.00%	27.05%	72.73%	83.33%
	(0/286)	(0/11)	(0/13)	(76/281)	(8/11)	(20/24)
sha256_arm_shani.cpp	✗	✗	✗	22.22%	50.00%	30.92%
				(292/1314)	(1/2)	(64/207)

Table 32: Coverage comparison between OSS-Fuzz and DeltAFLy

From the above table, it can be seen that by leveraging DeltAFLy and differential fuzzing, a higher code coverage was achieved than OSS-Fuzz's mutation-based fuzzer, particularly for the SHA-NI implementations on both `x86_64` and `AArch64` CPU architectures.

Success

The fuzzing campaign did not reveal any bugs or discrepancies between Bitcoin Core's and OpenSSL's implementations

The OSS-Fuzz coverage reference was taken from the monthly report on 30/07/2025.[33]

Info

The coverage measured in August and September 2025 exhibits substantial differences. The only constant is the Aarch64 file, which remains uncovered. The cause of these variations could not be determined.

[33] OSS-Fuzz Coverage Report from 30/07/2025

Info

DeltAFLy's coverage for the sha256.cpp file is lower than that of OSS-Fuzz because its harness targets only cryptographic primitives also implemented in OpenSSL. Several functions and regions within this file are specific to Bitcoin Core's custom implementations and, therefore, were not exercised during testing.

11.4. Getting Further

Cross-implementation differential fuzzing

Bitcoin Core's logic has been reimplemented in several other programming languages, including Go [59], Rust [60], Scala [61], and C# [62]. These alternative implementations also include cryptographic primitives required by the protocol [63].

Performing cross-implementation differential fuzzing was initially considered for this assessment but ultimately excluded, as ongoing community efforts already cover this area extensively. In particular, bitcoinfuzz [22], an actively maintained framework interfacing with most of the aforementioned projects, continuously performs differential testing across multiple Bitcoin implementations. This initiative has already uncovered a non-negligible number of issues, the vast majority of which were found in the alternative implementations rather than in Bitcoin Core itself. Since those external projects fall outside the scope of this engagement, further cross-implementation testing was not pursued.

Version-Based Differential Fuzzing

Version-based differential fuzzing aims to identify regressions by comparing the behavior of different versions of the same component under identical fuzzing inputs. However, applying this approach broadly within Bitcoin Core remains challenging due to the system's largely monolithic architecture. While emerging modular components such as `libbitcoinkernel` [11] could eventually make version-based differential fuzzing more practical, they currently account for only a small fraction of the overall codebase.

As a result, strict version differential fuzzing was not performed as part of this assessment. Instead, potential regressions were examined in a more lightweight manner, as described in Section 12.

12. Non-Regression Differential Testing

Differential testing allows comparing different implementations of the same algorithm, or even different versions of a single implementation that are expected to behave identically. The main challenge lies in defining what "behavior" to compare. This can be done from a control flow perspective (i.e., which functions are called and in what order) or a data perspective (i.e., what outputs or objects are produced). Both can vary across versions even if the algorithm performs the same task.

Differential fuzzing, as discussed in Section 11, is particularly suited for well-defined algorithms such as cryptographic primitives, where inputs and outputs can be compared straightforwardly. However, extending this to broader code areas becomes more challenging.

Current testing frameworks (unit, fuzzing, integration) do not fully capture regressions or behavioral changes across versions, leaving a blind spot in the test toolkit. Differential fuzzing partially addresses this issue but is mostly limited to narrowly scoped components.

To broaden coverage, we experimented with a **data-oriented** non-regression testing approach. The underlying idea is that a given input, when run through a harness, should produce the same side effects on the system (e.g., mempool, UTXO set, wallet state) across different versions. The goal is to generate a data-oriented trace of these side effects, which can then be compared across commits, releases, or versions. To implement this, the tracepoints mechanism called USDT (Userland Statically Defined Tracing) is particularly useful.

Info

This is a testing approach, not a fuzzing approach, as no input mutation or selection is performed.

12.1. Tracepoints (USDT)

User-space Statically Defined Tracing (USDT) is a mechanism that allows inserting tracepoints in the code to transmit live runtime information from the running node to external tools[34]. These tracepoints can feed an eBPF program running in the kernel, from which the data can be retrieved in userland via a buffer mechanism.

This mechanism is primarily intended for external monitoring and observability of the node. It can capture various events related to peers, the mempool, or the UTXO cache. Table 33 provides an overview of all available tracepoints.

[34] https://github.com/bitcoin/bitcoin/blob/master/doc/tracing.md

Tracepoint
net:inbound_message
net:outbound_message
net:inbound_connection
net:outbound_connection
net:evicted_inbound_connection
net:misbehaving_connection
net:closed_connection
validation:block_connected
utxocache:flush
utxocache:add
utxocache:spent
utxocache:uncache
coin_selection:selected_coins
coin_selection:normal_create_tx_internal
coin_selection:attempting_aps_create_tx
coin_selection:aps_create_tx_internal
mempool:added
mempool:removed
mempool:replaced
mempool:rejected

Table 33: Tracepoints list

Each tracepoint provides parameters containing data about the associated event, as illustrated in the example below. In this case, the event is triggered for every inbound connection accepted by the node.

```
TRACEPOINT(net, inbound_connection,
    pnode->GetId(),
    pnode->m_addr_name.c_str(),
    pnode->ConnectionTypeAsString().c_str(),
    pnode->ConnectedThroughNetwork(),
    GetNodeCount(ConnectionDirection::In));
```

12.2. Data-oriented Differential Testing (for Non-regression)

To perform this testing, we reuse the fuzzing target `fuzz`, compiled with USDT enabled (`-DWITH_USDT=ON`) so that tracepoints are active. This allows replaying the input corpus while capturing the output using a monitoring script that reads the eBPF buffers. Initial experiments attempted to capture tracepoint data through eBPF. However, the event throughput proved

too high, making it impossible to catch all events unless sending was throttled on the harness side. This approach slowed down testing and was found to be fragile and unreliable.

As a result, we modified the `TRACEPOINT` macro to directly write tracepoint data to a file instead of sending it to eBPF. The file is read using an environment variable, similarly to LLVM Coverage. The codebase provides the `AutoFile` utility class to serialize arbitrary data to a file. The environment variable used for the output file is `FUZZ_DATA_TRACE_FILE`. The following snippet illustrates the modified macro behavior.

```
template <typename T>
bool WriteToDataStream(const T& obj) noexcept
{
#if defined(ENABLE_DIFF_TRACING)
    if (!g_datatrace_stream || g_datatrace_stream->IsNull()) {
        return false;
    }
    try {
        *g_datatrace_stream << obj;
        return true;
    } catch (const std::exception&) {
        return false;
    }
#else
    return true;
#endif
}

template <typename T, typename... Args>
bool WriteToDataStream(const T& obj, const Args&... args) noexcept
{
  bool res = WriteToDataStream(obj);
  res &= WriteToDataStream(args...);
  return res;
}

#define SERIALIZE_TO_DATATRACE(...) WriteToDataStream(__VA_ARGS__)

#define TRACEPOINT(context, event, ...)                                    \
    do {                                                                   \
        SERIALIZE_TO_DATATRACE( #context , #event __VA_OPT__(, ) __VA_ARGS__); \
    } while(0)
```

As shown in the snippet, the macro `SERIALIZE_TO_DATATRACE` is also defined, allowing harnesses to serialize any additional data to the trace, expanding the data surface beyond existing tracepoints.

Testing is performed using a runner script that, for each harness and each input in the corpus, executes the following steps:

- Set the environment variable `FUZZ_DATA_TRACE_FILE` to a temporary file.
- Run the first version of the `fuzz` target.

- Update the environment variable and run the second version of the `fuzz` target.
- Hash the two output files and compare the resulting hashes.

Three outcomes are possible:

- Both files are empty, indicating no tracepoints were hit. Nothing can be concluded.
- The files are identical, meaning no behavioral change occurred.
- The files differ, indicating a behavioral change that should be investigated.

The macro also serializes the `TRACEPOINT` name within the file, making it easier to debug differences. An example of the hexadecimal output is shown in Figure 26.

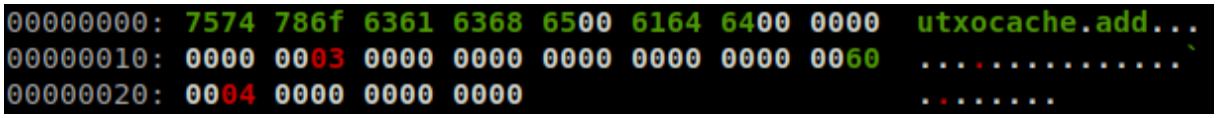


Figure 26: Hexadecimal datatrace output example

12.3. Results

Testing was performed on the following Bitcoin Core versions, though the approach is portable to any release:

- 29.0rc1 (commit `e9e6825b8cbe7b98ee07d068b78f84597e3a9652`)
- 29.0 (commit `f490f5562d4b20857ef8d042c050763795fd43da`)
- 29.1 (commit `fd784f277427aea7b25a8cdcd328b18a9fa64c0d`)

For these experiments, no additional `SERIALIZE_TO_DATATRACE` calls were inserted into the harnesses due to time constraints. As a result, the outcomes are solely based on data produced by the `TRACEPOINT` macro. The comparison between versions `29.0rc1` and `29.1` is summarized in Table 34. Only harnesses that produced non-empty traces are shown. All others generated empty traces.

Harness	OK	KO	Empty
connman	233	0	1464
coinscache_sim	217	0	12
coins_view	1369	0	490
ephemeral_package_eval	839	0	8
mini_miner_selection	454	0	44
mini_miner	601	0	0
net	328	0	683
p2p_handshake	731	0	0
p2p_headers_presync	618	0	0
process_message	2335	0	0
psbt	152	0	3744
partially_downloaded_block	390	0	89
process_messages	3103	0	0
package_rbf	483	0	105
rbf	425	0	61
rpc	168	94	7507
tx_pool_standard	918	0	0
tx_package_eval	964	0	17
tx_pool	2986	0	0
utxo_total_supply	0	815	0
validation_load_mempool	871	0	127
wallet_create_transaction	1067	0	82
wallet_notifications	906	0	30

Table 34: Differential test result v29.0rc1 vs v29.0

As the results show, only 23 harnesses out of 220+ produced non-empty traces. All harnesses are data-wise equivalent except for `rpc` and `utxo_total_supply`. Manual analysis revealed that the `utxocache::Flush` tracepoint is the source of all observed differences. Its first argument represents the time spent flushing the cache in microseconds, which is inherently non-deterministic and thus causes discrepancies between traces.

```
TRACEPOINT(utxocache, flush,
    int64_t{Ticks<std::chrono::microseconds>(NodeClock::now() - nNow)},
    (uint32_t)mode,
    (uint64_t)coins_count,
    (uint64_t)coins_mem_usage,
    (bool)fFlushForPrune);
```

Besides, the tracepoints show that both versions behave identically on these harnesses. Notably, the results between `29.0` and `29.1` are identical.

12.4. Limitations and Future Work

This approach could be a valuable addition to the existing set of testing methodologies. However, as the results indicate, reducing noise in the data trace requires that the submitted data be deterministic. For instance, it would be useful to modify the `utxocache::flush` tracepoint to either remove its first parameter or adjust its behavior when `datatrace` is enabled.

Additionally, in its current form, this testing approach is limited to harnesses that trigger tracepoints. The `SERIALIZE_TO_DATATRACE` macro, however, allows serialization of any data at any location in the code. Leveraging this macro more extensively could help capture otherwise untested computations, thereby broadening the coverage and usefulness of this data-oriented non-regression testing.

13. Conclusion

13.1. State of Testing

Testing measures (unit tests, integration tests, and fuzzing) are generally robust and cover a large portion of the codebase. Although fuzzing is not using structured mutation *per-se*, it nonetheless manages to exercise a wide range of branches. Bitcoin Core, however, is a complex, highly stateful application: it maintains pending transactions, active peer connections, a UTXO set, and the full blockchain. Most existing harnesses target narrow, well-scoped portions of the code in isolation.

Exercising the entire system in a sound, reproducible way requires expensive initialization and cleanup between iterations. Results from Section 8.4 and Section 8.5 demonstrate that performance degrades rapidly under such heavy setup/teardown costs. The most promising way to advance coverage in these areas is to adopt snapshot fuzzing.

Snapshot fuzzing captures a process snapshot (memory and registers) and restores it before each iteration so the fuzzer restarts from a fresh, pre-initialized state. This eliminates the need to re-run complex initialization and cleanup logic on every run and delegates that burden to the snapshot engine.

Snapshot fuzzing has not been explored in this assessment, as Niklas Goegge is actively developing Fuzzamoto [20], a custom snapshot fuzzer built on LibAFL and the Nyx engine [64]. Tailored specifically for Bitcoin Core, it leverages a custom intermediate representation ([65]) to enable grammar-based mutations. Fuzzamoto has already begun uncovering bugs in the code and in merge requests[35]. Accordingly, the security assessment focused on other approaches.

In our view, the most promising directions for improving Bitcoin Core testing are:

- Snapshot fuzzing with Fuzzamoto, which combines optimal design choices for stateful fuzzing
- Differential fuzzing to detect deviations between implementations or versions
- Data-oriented testing (see Section 12), which can complement property-based testing derived from assertions in existing harnesses

Finally, formal methods such as symbolic execution could be applied to carefully selected components, such as the Bitcoin Script interpreter, to enhance verification and testing. This approach was not pursued due to time constraints.

13.2. Findings & Contributions

This security assessment provided an independent, comprehensive review of the Bitcoin Core project, including its source code and auxiliary utilities, with the goal of evaluating and enhancing the overall quality of the codebase. No significant security issues were identified. Most recommendations focus on refining existing fuzzing harnesses to further improve their effectiveness and coverage.

[35] https://github.com/bitcoin/bitcoin/pull/30277#issuecomment-2992101654

The assessment also offered an opportunity to explore additional testing methodologies—such as structured fuzzing, differential fuzzing, and other complementary approaches—that can enhance the robustness of Bitcoin Core's testing ecosystem.

As part of the deliverables, all fuzzing corpora, scripts, and Dockerfiles have been provided to Brink to enable reproducibility and facilitate integration of techniques like ensemble fuzzing into their testing pipeline.

Finally, the newly developed fuzzing harnesses will be submitted directly as merge requests to the main Bitcoin Core repository for upstream inclusion.

Appendix

A. Unit-Test Suite

The table below lists the unit tests suite available at the time of the audit. Tests are grouped by components.

Test Suite	#Test
addrman_tests	23
allocator_tests	3
amount_tests	4
argsman_tests	14
arith_uint256_tests	13
banman_tests	1
base32_tests	2
base58_tests	2
base64_tests	2
bech32_tests	4
bip32_tests	6
bip324_tests	1
blockchain_tests	7
blockencodings_tests	8
blockfilter_index_tests	2
blockfilter_tests	5
blockmanager_tests	4
bloom_tests	12
bswap_tests	1
checkqueue_tests	10
cluster_linearize_tests	1
coins_tests	9
coinscachepair_tests	4
coinstatsindex_tests	2
common_url_tests	4
compilerbug_tests	1
compress_tests	6
crypto_tests	17
cuckoocache_tests	5
dbwrapper_tests	9
denialofservice_tests	5
descriptor_tests	1
disconnected_transactions	1
feefrac_tests	1
flatfile_tests	4
fs_tests	5
getarg_tests	8
hash_tests	2

Test Suite	#Test
headers_sync_chainwork_tests	1
httpserver_tests	1
i2p_tests	3
interfaces_tests	6
key_io_tests	3
key_tests	7
logging_tests	7
mempool_tests	6
merkle_tests	6
merkleblock_tests	2
miner_tests	1
miniminer_tests	5
miniscript_tests	1
minisketch_tests	1
multisig_tests	3
net_peer_connection_tests	1
net_peer_eviction_tests	2
net_tests	16
netbase_tests	15
node_warnings_tests	1
orphanage_tests	7
pcp_tests	10
peerman_tests	1
pmt_tests	2
policy_fee_tests	1
policyestimator_tests	1
pool_tests	4
pow_tests	15
prevector_tests	1
raii_event_tests	2
random_tests	8
rbf_tests	4
rest_tests	1
result_tests	2
reverselock_tests	3
rpc_tests	14
sanity_tests	1
scheduler_tests	4

Test Suite	#Test
script_p2sh_tests	6
script_parse_tests	1
script_segwit_tests	12
script_standard_tests	7
script_tests	19
scriptnum_tests	2
serfloat_tests	2
serialize_tests	13
settings_tests	4
sighash_tests	2
sigopcount_tests	2
skiplist_tests	4
sock_tests	6
span_tests	1
streams_tests	13
sync_tests	2
system_tests	1
testnet4_miner_tests	1
timeoffsets_tests	2
torcontrol_tests	2
transaction_tests	8
translation_tests	1
txdownload_tests	2
txindex_tests	1
txpackage_tests	9
txreconciliation_tests	3
txrequest_tests	1
txvalidation_tests	3
txvalidationcache_tests	2
uint256_tests	7
util_string_tests	1
util_tests	51
util_threadnames_tests	1
util_trace_tests	3
validation_block_tests	3
validation_chainstate_tests	2
validation_[...]_tests	8
validation_flush_tests	1
validation_tests	5
validationinterface_tests	2
versionbits_tests	2
db_tests	6
coinselector_tests	13
feebumper_tests	1
group_outputs_tests	1
init_tests	7

Test Suite	#Test
ismine_tests	1
psbt_wallet_tests	2
scriptpubkeyman_tests	2
spend_tests	2
wallet_crypto_tests	3
wallet_tests	17
wallet_transaction_tests	1
walletdb_tests	2
walletload_tests	2

B. Attack Class

Various attacks and threat scenarios considered throughout the audit.

B1. Memory Corruptions

This class of vulnerabilities includes all types of memory corruption issues such as buffer overflows, use-after-free, double free, heap overflows and stack overflows. Being developed in C++, Bitcoin Core is potentially exposed to these types of vulnerabilities.

B2. Denial-of-Service / Censorship

Denial of Service is the broad class of attacks preventing a system from providing the expected service. In the context of Bitcoin Core, it can be a node being prevented from receiving or sending blocks and transactions, or a wallet being prevented from broadcasting transactions. This can happen at various levels:

- network level: the node is prevented from communicating with other peers
- application level: the node is running but is prevented from processing blocks or transactions by exhausting its resources (CPU, memory, disk space). E.g: filling the mempool with junk transactions would prevent legitimate users from submitting their transactions.

B3. Network Partitioning

Network partitioning attacks aim to isolate a node or a group of nodes from the rest of the network, disrupting their ability to communicate and synchronize with other nodes. This can lead to various issues, including double-spending, censorship, and loss of consensus. Two notable types of network partitioning attacks are:

- **Erebus Attack**: An Erebus attack is a network-level attack where an adversary uses a small number of IP prefixes to partition and isolate Bitcoin nodes, controlling their view of the network and potentially enabling censorship. This attack exploits the structure of the internet's routing system, specifically targeting Autonomous Systems (AS) to manipulate the flow of information between nodes.
- **Eclipse Attack**: An Eclipse attack is a type of network attack where an adversary manages to isolate a target node from honest peers and keeps it connected to malicious peers. This can lead to censorship, double-spending, or other malicious activities [66].

B4. Consensus Attacks / Policy

Consensus attacks target the core principles of the Bitcoin network aiming to disrupt the agreement among nodes on the state of the blockchain. These attacks can lead to double-spending, chain reorganizations, censorship and ultimately loss of trust in the network. This class also encompass any discrepancy between the consensus rules and the policy rules that could be exploited by an attacker. A related attack is Timewrap Attack[36].

[36] https://bitcoin.stackexchange.com/questions/75831/what-is-time-warp-attack-and-how-does-it-work-in-general/75834#75834

C. Baseline Fuzzing Coverage

Harness	#corpus	#funcs	#region	#lines	#inst	#branches	#mcdc
addition_overflow	59	21	139	150	70	54	10
addr_info_deserialize	109	145	494	731	230	190	9
address_deserialize	130	191	609	948	300	258	8
addrman	1321	433	2327	3147	695	1069	127
addrman_serdeser	963	398	1872	2659	607	847	88
asmap	204	81	450	464	92	308	62
asmap_direct	177	13	151	157	14	85	6
autofile	234	61	186	260	83	80	1
banman	1075	362	2018	3036	449	1112	119
base32_encode_decode	34	16	74	104	18	37	8
base58_encode_decode	90	15	106	115	16	71	16
base58check_encode_decode	97	47	167	330	53	86	19
base64_encode_decode	24	14	56	89	16	23	3
bech32_random_decode	76	18	142	137	19	90	13
bech32_roundtrip	44	18	126	134	19	71	2
bip324_cipher_roundtrip	1445	249	2001	3436	253	381	5
bip324_ecdh	1507	163	1690	2544	164	270	3
bitdeque	587	94	222	467	106	83	5
bitset	1012	85	291	359	499	74	0
block	807	266	832	1596	535	380	36
block_deserialize	136	199	436	821	399	115	6
block_file_info_deserialize	31	73	168	278	97	35	1
block_filter_deserialize	58	92	236	401	109	69	2
block_header	109	114	311	639	137	118	6
block_header_and_short_txids_deserialize	157	213	469	872	470	123	7
block_index	334	687	2734	4455	795	946	26
blockfilter	393	178	575	1237	195	191	6
blockheader_deserialize	12	78	148	258	84	23	1
blocklocator_deserialize	32	94	203	358	108	48	2
blockmerkleroot	155	210	475	1110	410	136	6
blocktransactions_deserialize	149	192	428	795	383	115	6
blocktransactionsrequest_deserialize	46	89	210	357	99	58	3
blockundo_deserialize	176	202	772	1372	307	273	35
bloom_filter	604	216	743	1137	335	366	69
bloomfilter_deserialize	28	83	193	331	86	48	2
buffered_file	188	44	161	235	55	72	2
build_and_compare_feerate_diagram	70	18	77	80	20	28	8
chacha20_split_crypt	282	31	111	354	33	34	0
chacha20_split_keystream	357	22	99	317	23	35	0
chain	221	126	384	665	152	124	4
checkqueue	60	54	268	346	56	105	2
clusterlin_ancestor_finder	82	80	330	388	95	90	2
clusterlin_chunking	64	69	259	320	84	71	4
clusterlin_components	75	66	259	312	81	69	2
clusterlin_depgraph_serialization	111	84	422	486	121	155	2
clusterlin_depgraph_sim	145	87	422	485	116	153	3
clusterlin_fix_linearization	176	64	265	314	79	72	2
clusterlin_linearization_chunking	97	83	336	397	98	101	8
clusterlin_linearize	291	148	747	964	167	258	15
clusterlin_make_connected	122	89	440	509	118	160	2
clusterlin_merge	127	88	396	470	103	120	9
clusterlin_postlinearize	74	87	419	504	102	130	7
clusterlin_postlinearize_moved_leaf	75	77	359	441	92	112	5

Harness	#corpus	#funcs	#region	#lines	#inst	#branches	#mcdc
clusterlin_postlinearize_tree	258	147	780	1027	165	267	13
clusterlin_search_finder	229	130	588	777	148	181	6
coin_grinder	538	175	564	1031	193	225	32
coin_grinder_is_optimal	374	145	401	707	163	123	11
coincontrol	190	255	708	1044	272	205	8
coins_deserialize	68	180	703	1239	248	242	33
coins_view	1859	526	2609	3991	772	1410	175
coinscache_sim	229	150	501	778	161	184	20
coinselection_bnb	252	270	835	1587	313	292	20
coinselection_knapsack	287	285	923	1699	328	342	32
coinselection_srd	205	270	792	1546	313	262	14
connman	1697	602	3271	4185	843	1414	190
crypter	354	446	2786	4707	469	642	28
crypto	342	115	245	1045	119	70	10
crypto_aeadchacha20poly1305	395	52	195	602	56	66	0
crypto_aes256	57	30	175	426	31	56	1
crypto_aes256cbc	90	38	231	496	39	87	5
crypto_chacha20	571	24	102	325	25	36	0
crypto_common	18	33	57	122	34	1	0
crypto_diff_fuzz_chacha20	463	25	98	315	26	32	0
crypto_fschacha20	295	25	109	330	26	35	0
crypto_fschacha20poly1305	542	57	205	623	61	65	0
crypto_hkdf_hmac_sha256_l32	154	31	68	224	32	18	2
crypto_poly1305	29	19	64	200	20	18	0
crypto_poly1305_split	49	19	76	214	20	25	0
cuckoocache	113	22	73	140	23	33	2
data_stream_addr_man	1002	353	1601	2393	522	788	106
decode_tx	367	189	509	887	281	195	21
descriptor_parse	2224	816	5987	8265	1047	3415	379
diskblockindex_deserialize	34	90	208	325	134	43	1
ellswift_roundtrip	1754	227	2219	3323	228	460	18
ephemeral_package_eval	847	880	4433	5596	1209	1588	61
eval_script	1755	193	1259	1866	201	833	114
fee_rate	13	33	162	204	37	82	4
fee_rate_deserialize	10	69	139	232	72	23	1
feefrac	27	25	72	99	26	25	0
feefrac_div_fallback	73	28	123	155	30	60	7
feefrac_mul_div	79	39	176	218	42	82	12
fees	104	44	122	273	45	30	3
flat_file_pos_deserialize	20	74	165	273	94	37	1
flatfile	29	47	182	255	59	78	1
float	18	9	53	61	10	23	0
golomb_rice	175	130	473	949	135	163	3
headers_sync_state	193	257	984	1619	275	322	22
hex	230	242	777	1510	368	294	36
http_request	46	67	269	383	68	135	2
i2p	270	257	1221	1850	333	458	29
integer	333	195	840	1279	257	370	43
inv_deserialize	11	75	145	249	81	23	1
key	1172	550	3258	5282	588	860	42
key_io	297	150	588	1190	155	222	25
key_origin_info_deserialize	31	92	204	357	99	52	2
kitchen_sink	36	30	131	201	32	100	2
load_external_block_file	292	455	2008	2660	712	768	40
local_address	758	273	1116	1584	332	566	85
locale	33	26	152	193	36	80	5

Harness	#corpus	#funcs	#region	#lines	#inst	#branches	#mcdc
merkle_block_deserialize	61	111	239	421	152	58	2
merkleblock	313	201	496	901	337	164	17
message	2450	362	2588	4141	392	731	67
messageheader_deserialize	49	75	160	255	84	35	3
mini_miner	601	448	1484	2240	569	405	14
mini_miner_selection	498	557	1911	3073	733	547	16
miniscript_script	598	170	1565	1728	277	1177	141
miniscript_smart	1472	506	4221	6072	675	2519	200
miniscript_stable	1433	505	4226	6071	674	2537	200
miniscript_string	634	129	1359	1592	149	1060	99
minisketch	298	129	416	671	263	183	9
mocked_descriptor_parse	2527	822	6029	8320	1055	3454	392
muhash	737	83	289	707	87	92	2
multiplication_overflow	62	6	15	20	7	1	0
natpmp_request_port_map	107	152	695	953	198	263	13
net	1011	306	1267	1732	411	518	79
net_permissions	308	136	734	1017	156	414	52
netaddr_deserialize	87	153	508	779	173	232	6
netaddress	389	270	1148	1965	328	685	93
netbase_dns_lookup	493	159	653	1092	178	348	56
node_eviction	277	29	113	170	31	60	7
num3072_inv	116	50	302	487	52	133	5
num3072_mul	90	42	210	288	43	102	5
out_point_deserialize	12	81	155	252	87	25	1
overflow	38	11	61	54	37	33	10
p2p_handshake	731	656	3390	4101	865	1252	142
p2p_headers_presync	618	786	4029	4750	1454	1273	35
p2p_transport_bidirectional	337	149	434	682	161	91	12
p2p_transport_bidirectional_v1v2	743	276	1832	2546	290	338	15
p2p_transport_bidirectional_v2	1363	320	2542	3901	325	548	17
p2p_transport_serialization	269	139	441	645	150	92	10
package_rbf	588	434	1412	2301	629	461	21
parse_hd_keypath	56	27	177	224	28	93	1
parse_iso8601	30	27	170	206	40	90	15
parse_numbers	108	24	240	242	36	161	45
parse_script	174	52	312	391	57	329	22
parse_univalue	2101	603	4652	6578	806	2825	354
partial_merkle_tree_deserialize	57	103	229	402	122	58	2
partially_downloaded_block	479	486	1507	2375	726	425	10
partially_signed_transaction_deserialize	1592	364	1369	2694	839	672	49
pcp_request_port_map	136	167	832	1123	225	347	23
policy_estimator	526	296	1074	1531	520	365	13
policy_estimator_io	155	117	528	694	156	210	9
pool_resource	464	31	122	190	110	46	2
pow	203	79	306	417	83	151	22
pow_transition	118	49	234	324	50	105	5
prefilled_transaction_deserialize	122	193	434	804	392	119	7
prevector	160	85	181	340	95	58	0
primitives_transaction	367	196	448	872	303	150	11
process_message	2335	1308	8780	10033	2560	3683	383
process_messages	3103	1333	9086	10352	2625	3923	450
protocol	22	54	220	314	68	107	5
psbt	3896	726	4573	7226	1088	2480	306
psbt_base64_decode	1224	300	1199	2369	493	579	52
psbt_input_deserialize	737	322	1068	2194	738	454	34
psbt_output_deserialize	365	230	792	1523	324	308	29

Harness	#corpus	#funcs	#region	#lines	#inst	#branches	#mcdc
pub_key_deserialize	38	72	192	308	76	62	8
random	832	44	118	269	48	26	0
rbf	486	387	1142	1936	577	314	8
rolling_bloom_filter	162	109	381	849	115	137	2
rpc	7769	2496	17882	30764	3851	9240	872
script	2119	476	2352	3594	554	1458	183
script_descriptor_cache	117	65	267	607	66	84	13
script_deserialize	24	99	234	401	102	64	2
script_flags	2088	492	3745	5358	641	1712	246
script_format	2068	322	1539	2415	358	943	99
script_interpreter	410	207	611	1050	343	258	27
script_ops	182	83	309	437	89	161	26
script_parsing	34	11	61	68	12	37	15
script_sigcache	471	404	2259	3754	519	537	46
script_sign	3503	766	5403	7544	1067	2631	278
scriptnum_ops	84	36	95	95	37	34	0
scriptpubkeyman	4808	1419	10125	13494	2014	5276	524
secp256k1_ec_seckey_import_export_der	66	111	1006	1454	112	125	10
secp256k1_ecdsa_signature_parse_der_lax	66	201	1343	2266	206	277	16
service_deserialize	92	172	542	850	204	236	6
signature_checker	1897	207	1459	2049	217	967	140
signet	1431	495	2602	4756	695	1159	98
snapshotmetadata_deserialize	17	249	639	1403	332	187	3
socks5	41	78	411	461	107	146	4
span	4	6	15	20	7	1	0
str_printf	442	34	252	358	144	171	24
string	633	172	1129	1501	239	740	109
system	388	131	742	905	154	382	34
timeoffsets	142	77	295	384	84	95	0
torcontrol	238	149	760	904	165	350	36
transaction	950	590	2386	3971	988	1289	122
tx_in	65	136	371	585	180	134	4
tx_in_deserialize	44	127	281	454	155	73	2
tx_out	38	120	381	574	136	161	12
tx_package_eval	981	910	4775	6024	1266	1942	109
tx_pool	2986	1190	7361	10017	1593	3012	277
tx_pool_standard	918	976	4911	6402	1351	1846	75
txdownloadman	498	493	1718	2641	648	635	54
txdownloadman_impl	528	483	1777	2668	638	665	56
txgraph	2258	348	2177	2876	378	851	51
txorphan	283	211	637	1072	311	194	4
txoutcompressor_deserialize	65	175	690	1219	238	241	33
txrequest	221	90	384	426	100	203	42
txundo_deserialize	156	197	763	1359	281	273	35
uint160_deserialize	10	60	127	204	63	23	1
uint256_deserialize	9	60	127	204	63	23	1
utxo_snapshot	402	1253	6217	8751	2019	2073	85
utxo_snapshot_invalid	175	975	4034	6619	1433	1325	59
utxo_total_supply	815	1650	8979	12510	2706	3008	116
validation_load_mempool	998	608	2475	3452	873	890	89
vecdeque	269	45	186	197	284	63	2
versionbits	139	26	149	212	28	83	6
wallet_bdb_parser	48	264	1003	1436	320	382	30
wallet_create_transaction	1149	1296	7568	10365	1694	2892	139
wallet_fees	23	161	547	671	177	161	4
wallet_notifications	936	1261	6123	9164	1703	2204	107

D. Elligator Swift Non Distinguishability

```
#include <iostream>
#include <fstream>
#include <vector>
#include <iomanip>
#include <cstring>
#include <random>

#include <secp256k1.h>
#include <secp256k1_ellswift.h>

void fill_random(unsigned char* buffer, size_t size) {
    std::random_device rd;
    for (size_t i = 0; i < size; i++) {
        buffer[i] = rd() & 0xFF;
    }
}

int main() {
    // Create secp256k1 context
    secp256k1_context* ctx = secp256k1_context_create(SECP256K1_CONTEXT_NONE);
    if (!ctx) {
        std::cerr << "Failed to create secp256k1 context" << std::endl;
        return 1;
    }

    // Generate random seed for private key
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<unsigned char> dis(0, 255);

    unsigned char privkey[32];
    unsigned char aux_rand[32];
    unsigned char ellswift_pubkey[64];

    // Open the ElligatorSwift encoded public key to a file
    std::ofstream pubkey_file("ellswift_pubkeys_stream.bin", std::ios::binary);
    if (!pubkey_file) {
        std::cerr << "Failed to open public key file for writing" << std::endl;
        secp256k1_context_destroy(ctx);
        return 1;
    }

    for(int i = 0; i < 4096; i++) {

        fill_random(privkey, 32);
        fill_random(aux_rand, 32);

        // Verify the private key is valid
        if (!secp256k1_ec_seckey_verify(ctx, privkey)) {
```

```
            std::cerr << "Generated private key is invalid" << std::endl;
            secp256k1_context_destroy(ctx);
            return 1;
        }

        // Create ElligatorSwift encoded public key
        if (!secp256k1_ellswift_create(ctx, ellswift_pubkey, privkey, aux_rand)) {
            std::cerr << "Failed to create ElligatorSwift public key" << std::endl;
            secp256k1_context_destroy(ctx);
            return 1;
        }

        pubkey_file.write(reinterpret_cast<const char*>(ellswift_pubkey), 64);

    }

    pubkey_file.close();
    secp256k1_context_destroy(ctx);
    return 0;
}
```

E. Miniscript Grammar

```
{
    "<start>": [["<miniscript>"]],

    "<type>": [["\u0000"], ["\u0001"]],
    "<bool>": [["0"], ["1"]],
    "<onenine>": [["1"], ["2"], ["3"], ["4"], ["5"], ["6"], ["7"], ["8"], ["9"]],
    "<1-9>": [["<onenine>"]],
    "<digit>": [["0"], ["<onenine>"]],
    "<digit-1>": [["<digit>"], ["<digit>", "<digit-1>"]],
    "<digits>": [["<digit-1>"]],
    "<pos-digits-infinite>": [["<onenine>"], ["<onenine>", "<pos-digits>"]],
    "<pos-digits-limited>": [["<1-9>"], ["<1-9>", "<1-9>"], ["<1-9>", "<1-9>", "<1-9>"], ["<1-9>", "<1-9>", "<1-9>", "<1-9>"],
    ["<1-9>", "<1-9>", "<1-9>", "<1-9>", "<1-9>"],
                            ["<1-9>", "<1-9>", "<1-9>", "<1-9>", "<1-9>", "<1-9>"], ["<1-9>", "<1-9>", "<1-9>", "<1-9>", "<1-9>",
                            "<1-9>"], ["<1-9>", "<1-9>", "<1-9>", "<1-9>", "<1-9>", "<1-9>", "<1-9>"]],



    "<alpha>": [["A"],["B"],["C"],["D"],["E"],["F"],["G"],["H"],["I"],["J"],["K"],["L"],["M"],["N"],["O"],["P"],["Q"],["R"],["S"],
    ["T"],["U"],["V"],["W"],["X"],["Y"],["Z"],
                ["a"],["b"],["c"],["d"],["e"],["f"],["g"],["h"],["i"],["j"],["k"],["l"],["m"],["n"],["o"],["p"],["q"],["r"],["s"],
                ["t"],["u"],["v"],["w"],["x"],["y"],["z"]],
    "<pascii>": [["<alpha>"],["<digit>"],["\t"],["\n"],[" "],["!"],["\""],["#"],["$"],["%"],["&"],["'"],["("],[")"],["*"],
                        ["+"],[","],["-"],["."],["/"],[":"],[";"],["<"],["="],[">"],["?"],["@"],["["],["\\"],["]"],["^"],["_"],
                        ["`"],["{"],["|"],["}"],["~"]],
    "<p>": [["<alpha>"],["<digit>"]],

    "<byte>": [["<pascii>"],["\\00"],["\\01"],["\\02"],["\\03"],["\\04"],["\\05"],["\\06"],["\\07"],["\\08"],["\\0b"],["\\0c"],
    ["\r"],["\\0e"],["\\0f"],["\\10"],["\\11"],["\\12"],["\\13"],
               ["\\14"],["\\15"],["\\16"],["\\17"],["\\18"],["\\19"],["\\1a"],["\\1b"],["\\1c"],["\\1d"],["\\1e"],["\\1f"],["\\7f"],
               ["\\80"],
               ["\\81"],["\\82"],["\\83"],["\\84"],["\\85"],["\\86"],["\\87"],["\\88"],["\\89"],["\\8a"],["\\8b"],["\\8c"],["\\8d"],
               ["\\8e"],["\\8f"],["\\90"],["\\91"],["\\92"],["\\93"],
               ["\\94"],["\\95"],["\\96"],["\\97"],["\\98"],["\\99"],["\\9a"],["\\9b"],["\\9c"],["\\9d"],["\\9e"],["\\9f"],["\\a0"],
               ["\\a1"],["\\a2"],["\\a3"],["\\a4"],["\\a5"],["\\a6"],
               ["\\a7"],["\\a8"],["\\a9"],["\\aa"],["\\ab"],["\\ac"],["\\ad"],["\\ae"],["\\af"],["\\b0"],["\\b1"],["\\b2"],["\\b3"],
               ["\\b4"],["\\b5"],["\\b6"],["\\b7"],["\\b8"],["\\b9"],
               ["\\ba"],["\\bb"],["\\bc"],["\\bd"],["\\be"],["\\bf"],["\\c0"],["\\c1"],["\\c2"],["\\c3"],["\\c4"],["\\c5"],["\\c6"],
               ["\\c7"],["\\c8"],["\\c9"],["\\ca"],["\\cb"],["\\cc"],
               ["\\cd"],["\\ce"],["\\cf"],["\\d0"],["\\d1"],["\\d2"],["\\d3"],["\\d4"],["\\d5"],["\\d6"],["\\d7"],["\\d8"],["\\d9"],
               ["\\da"],["\\db"],["\\dc"],["\\dd"],["\\de"],["\\df"],
               ["\\e0"],["\\e1"],["\\e2"],["\\e3"],["\\e4"],["\\e5"],["\\e6"],["\\e7"],["\\e8"],["\\e9"],["\\ea"],["\\eb"],["\\ec"],
               ["\\ed"],["\\ee"],["\\ef"],["\\f0"],["\\f1"],["\\f2"],
               ["\\f3"],["\\f4"],["\\f5"],["\\f6"],["\\f7"],["\\f8"],["\\f9"],["\\fa"],["\\fb"],["\\fc"],["\\fd"],["\\fe"],["\
               \ff"]],



    "<hex>": [["<digit>"], ["a"], ["b"], ["c"], ["d"], ["e"], ["f"],
                           ["A"], ["B"], ["C"], ["D"], ["E"], ["F"]],
    "<hex-4-bytes>": [["<hex>","<hex>","<hex>","<hex>","<hex>","<hex>","<hex>","<hex>"]],
    "<hex-8-bytes>": [["<hex-4-bytes>", "<hex-4-bytes>"]],
    "<hex-20-bytes>": [["<hex-8-bytes>","<hex-8-bytes>","<hex-4-bytes>"]],
    "<hex-32-bytes>": [["<hex-8-bytes>","<hex-8-bytes>","<hex-8-bytes>","<hex-8-bytes>"]],
    "<hex-33-bytes>":[["<hex-32-bytes>","<hex>","<hex>"]],

    "<label>": [["<p>"], ["<p>", "<p>"], ["<p>", "<p>", "<p>"], ["<p>", "<p>", "<p>", "<p>"], ["<p>", "<p>", "<p>", "<p>", "<p>"],
    ["<p>", "<p>", "<p>", "<p>", "<p>", "<p>"]],

    "<tapkey>": [["<hex-32-bytes>"]],
    "<p2wshkey>": [["<hex-33-bytes>"]],

    "<key>": [["<tapkey>"], ["<p2wshkey>"]],
    "<keylabel>": [["<label>"]],

    "<check_1>": [["pk_k(", "<keylabel>", ")"]],
    "<check_2>": [["pk_h(", "<keylabel>", ")"]],
```

```
    "<check_3>": [["pk(", "<keylabel>", ")"]],
    "<check_4>": [["pkh(", "<keylabel>", ")"]],
    "<check>": [["<check_1>"], ["<check_2>"], ["<check_3>"], ["<check_4>"]],

    "<older>": [["older(", "<pos-digits-limited>", ")"]],
    "<after>": [["after(", "<pos-digits-limited>", ")"]],

    "<h-orhex32>": [["H"], ["<hex-32-bytes>"]],
    "<sha256>": [["sha256(", "<h-orhex32>", ")"]],
    "<hash256>": [["hash256(", "<h-orhex32>", ")"]],

    "<h-orhex20>": [["H"], ["<hex-20-bytes>"]],
    "<ripemd160>": [["ripemd160(", "<h-orhex20>", ")"]],
    "<hash160>": [["hash160(", "<h-orhex20>", ")"]],
    "<hashes>": [["<sha256>"], ["<hash256>"], ["<ripemd160>"], ["<hash160>"]],

    "<andorb>": [["andor(", "<BaseExpr>", ",", "<BaseExpr>", ",", "<BaseExpr>", ")"]],
    "<andork>": [["andor(", "<BaseExpr>", ",", "<KeyExpr>", ",", "<KeyExpr>", ")"]],
    "<andorv>": [["andor(", "<BaseExpr>", ",", "<VerifyExpr>", ",", "<VerifyExpr>", ")"]],
    "<andor>": [["<andorb>"],["<andork>"], ["<andorv>"]],


    "<and_vb>": [["and_v(", "<VerifyExpr>", ",", "<BaseExpr>", ")"]],
    "<and_vk>": [["and_v(", "<VerifyExpr>", ",", "<KeyExpr>", ")"]],
    "<and_vv>": [["and_v(", "<VerifyExpr>", ",", "<VerifyExpr>", ")"]],
    "<and_v>": [["<and_vb>"], ["<and_vk>"], ["<and_vv>"]],


    "<and_b>": [["and_b(", "<BaseExpr>", ",", "<WrapExpr>", ")"]],
    "<and_n>": [["and_n(", "<BaseExpr>", ",", "<BaseExpr>", ")"]],


    "<or_b>": [["or_b(", "<BaseExpr>", ",", "<WrapExpr>", ")"]],
    "<or_c>": [["or_c(", "<BaseExpr>", ",", "<VerifyExpr>", ")"]],
    "<or_d>": [["or_d(", "<BaseExpr>", ",", "<BaseExpr>", ")"]],

    "<or_ib>": [["or_i(", "<BaseExpr>", ",", "<BaseExpr>", ")"]],
    "<or_ik>": [["or_i(", "<KeyExpr>", ",", "<KeyExpr>", ")"]],
    "<or_iv>": [["or_i(", "<VerifyExpr>", ",", "<VerifyExpr>", ")"]],


    "<thresh-1>": [[",", "<thresh-1-1>"]],
    "<thresh-x>": [[], ["<thresh-1>"]],
    "<thresh-1-1>": [["<WrapExpr>", "<thresh-x>"]],
    "<thresh>": [["thresh(", "<digits>",",", "<BaseExpr>", "<thresh-1>", ")"]],

    "<kkeys-1>": [[",", "<kkeys-1-and-more>"]],
    "<kkeys-0-1>": [[], ["<kkeys-1>"]],
    "<kkeys-1-and-more>": [["<keylabel>", "<kkeys-0-1>"]],
    "<kkeys>": [["<digits>", "<kkeys-1>"]],
    "<multi-check-p2wsh>": [["multi(", "<kkeys>", ")"]],
    "<multi-check-tapscript>": [["multi_a(", "<kkeys>", ")"]],

    "<Type-BW>": [["a"],["s"]],
    "<Type-KB>": [["c"]],
    "<Type-VB>": [["d"], ["t"]],
    "<Type-BB>": [["j"], ["n"], ["l"], ["u"]],
    "<Type-BV>": [["v"]],

    "<Type-BW-Id>": [["<Type-BW>", ":", "<BaseExprNoW>"], ["<Type-BW>", "<Type-VB-Id>"], ["<Type-BW>", "<Type-BB-Id>"], ["<Type-
    BW>", "<Type-KB-Id>"]],
    "<Type-KB-Id>": [["<Type-KB>", ":", "<KeyExpr>"]],
    "<Type-VB-Id>": [["<Type-VB>", ":", "<VerifyExprNoW>"]],
    "<Type-BV-Id>": [["<Type-BV>", ":", "<BaseExprNoW>"], ["<Type-BV>", "<Type-KB-Id>"], ["<Type-BV>", "<Type-VB-Id>"], ["<Type-
    BV>", "<Type-BB-Id>"]],
    "<Type-BB-Id>": [["<Type-BB>", ":", "<BaseExprNoW>"], ["<Type-BB>", "<Type-KB-Id>"], ["<Type-BB>", "<Type-VB-Id>"], ["<Type-
    BB>", "<Type-BB-Id>"]],
```

```
    "<BaseExprWrappers>": [["<Type-KB-Id>"], ["<Type-VB-Id>"], ["<Type-BB-Id>"]],
    "<BaseExprNoW>": [["<bool>"], ["<older>"], ["<after>"], ["<hashes>"],
                    ["<and_b>"], ["<or_b>"], ["<or_d>"], ["<thresh>"],
                    ["<multi-check-p2wsh>"], ["<multi-check-tapscript>"],
                    ["<andorb>"], ["<and_n>"], ["<and_vb>"], ["<or_ib>"]],
    "<BaseExpr>": [["<BaseExprWrappers>"], ["<BaseExprNoW>"]],

    "<VerifyExprWrappers>": [["<Type-BV-Id>"]],
    "<VerifyExprNoW>": [["<or_c>"], ["<andorv>"], ["<and_vv>"], ["<or_iv>"]],
    "<VerifyExpr>": [["<VerifyExprWrappers>"],["<VerifyExprNoW>"]],

    "<KeyExpr>": [["<check>"], ["<andork>"], ["<and_vk>"], ["<or_ik>"]],

    "<WrapExpr>": [["<Type-BW-Id>"]],

    "<fragment>": [["<BaseExpr>"], ["<VerifyExpr>"], ["<KeyExpr>"], ["<WrapExpr>"]],

    "<miniscript>": [["<fragment>", "<type>"]]
}
```

F. ECDSA Signature Discrepancy

This appendix presents the discrepancy found between libsecp256k1 openssl for ECDSA signature check implementation.

Disclaimer

The function `ecdsa_signature_parse_der_lax` purposely do not check the sequence length in the ASN.1 signature encoding. This behavior is known[37] and mitigated by BIP66 which enforces strict DER encoding. The following snippets are for informational purposes and to better explain why a fuzzing harness was able to generate a seemingly valid ECDSA signature.

The `sig` is the signature, `pubkey_bytes` the public key in compressed format and `sighash` the message hash to be signed. The three values are originally taken from a fuzzing input which passed libsecp256k1 signature verification. The code snippet on the right, shows the equivalent ECDSA signature checks using OpenSSL. The `sig` array shows the DER breakdown of the signature. It first starts with a sequence tag followed by the length in bytes. As one can see it is invalid as the length here should be `0x06`.

libsecp256k1

```c
#include <secp256k1.h>

int main(void) {

  unsigned char sig[72] = {
    0x30, 0x02, 0x02, 0x01, 0x02, 0x02, 0x01, 0x02,
    0x41, 0x63, 0xc8, 0x00, 0x00, 0x00, 0x6c, 0xa8, // junk
    0x17, 0x57, 0x14, 0x7d, 0x63, 0x0a, 0x00, 0x31, // junk
    0x2d, 0x32, 0x32, 0xd5, 0x32, 0xff, 0xff, 0x74, // junk
    0xba, 0x00, 0x00, 0x00, 0xc8, 0xff, 0xff, 0xf9, // junk
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // junk
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // junk
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // junk
    0x00, 0x00, 0x00                                // junk
  };

  unsigned char pubkey_bytes[33] = {
    0x02, // COMPRESSED key
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02,
  };

  unsigned char sighash[32] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  };

  secp256k1_pubkey pubkey;
  secp256k1_ecdsa_signature ecdsa_sig;
```

OpenSSL

```c
#include <openssl/ecdsa.h>
int main() {

  unsigned char sig[] = {
    0x30, // Sequence tag
    0x02, // Length
    0x02, // Integer ID.
    0x01, // Length of R
    0x02, // Value
    0x02, // Integer ID.
    0x01, // Length of S
    0x02  // Value
  };

  unsigned char pubkey_bytes[33] = {
    0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02,
  };

  unsigned char sighash[32] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  };

  // Create EC_KEY from public key bytes
  EC_KEY *ec_key = EC_KEY_new_by_curve_name(NID_secp256k1);
  if (!ec_key) {
    fprintf(stderr, "EC_KEY_new_by_curve_name failed\n");
    return 1;
```

[37] discussed here and here.

```
  if (!secp256k1_ec_pubkey_parse(secp256k1_context_static,
  &pubkey, pubkey_bytes, sizeof(pubkey_bytes))) {
    printf("pubkey parse failed\n");
    return EXIT_FAILURE;
  }
  if (!ecdsa_signature_parse_der_lax(&ecdsa_sig, sig,
  sizeof(sig))) {
    printf("parse der lax failed\n");
    return EXIT_FAILURE;
  }

  secp256k1_ecdsa_signature_normalize(secp256k1_context_static,
  &ecdsa_sig, &ecdsa_sig);
  if(secp256k1_ecdsa_verify(secp256k1_context_static,
  &ecdsa_ecdsa_sig, sighash, &pubkey)) {
    printf("Signature is valid\n");
  } else {
    printf("Signature is NOT valid\n");
  }


  return EXIT_SUCCESS;
}
```

Result of compiling and running:

```
$ gcc -o test_ecdsa test_ecdsa.c -lsecp256k1
$ ./test_ecdsa
# Output: Signature is valid
```

```
  }

  const EC_GROUP *group = EC_KEY_get0_group(ec_key);
  EC_POINT *point = EC_POINT_new(group);

  // This can handle both compressed and uncompressed formats
  if (EC_POINT_oct2point(group, point, pubkey_bytes,
  sizeof(pubkey_bytes), NULL)) {
    EC_KEY_set_public_key(ec_key, point);
  }

  int ret = ECDSA_verify(0, sighash, sizeof(sighash), sig,
  sizeof(sig), ec_key);
  if (ret == 1) {
    printf("Signature is valid.\n");
  } else if (ret == 0) {
    printf("Signature is invalid.\n");
  } else {
    printf("Error verifying signature: %s\n",
    ERR_error_string(ERR_get_error(), NULL));
  }
  EC_KEY_free(ec_key);
}
```

Result of compiling and running:

```
$gcc -o test_openssl ecdsa_openssl.c -lcrypto
$ ./test_openssl
# Output: Signature is invalid.
```

G. Consume Transaction

The following function provides the logic to consume a transaction from the fuzzed input. It is used in the three added harnesses `ConnectBlock`, `ActivateBestChainStep` and `ActivateBestChain`.

```
CTransactionRef ConsumeTransaction(FuzzedDataProvider& fuzzed_data_provider,
                                   std::vector<CTxIn>& additionnalUTXO, bool coinbase=false,
                                   unsigned targetHeight=0) {

    CMutableTransaction tx;
    if (coinbase) {
        tx.vin.resize(1);
        tx.vin[0].prevout.SetNull();
        tx.vin[0].nSequence = CTxIn::MAX_SEQUENCE_NONFINAL; // Make sure timelock is enforced.
        if (fuzzed_data_provider.ConsumeBool()) {
            tx.vin[0].scriptSig = CScript() << targetHeight << OP_0;
        } else {
            auto scriptSig = ConsumeRandomLengthByteVector<unsigned char>(fuzzed_data_provider, 100);
            tx.vin[0].scriptSig = CScript(scriptSig.begin(), scriptSig.end());
        }
    } else {
        int numInput = fuzzed_data_provider.ConsumeIntegralInRange<int>(0, 10);
        tx.vin.resize(numInput);
        for (int i = 0; i < numInput; i++) {
            uint32_t targetUTXO = fuzzed_data_provider.ConsumeIntegralInRange<uint32_t>(0,
            allUTXO.size() + additionnalUTXO.size() - 1);
            if (targetUTXO < allUTXO.size()) {
                tx.vin[i] = allUTXO[targetUTXO];
            } else {
                Assert(targetUTXO - allUTXO.size() < additionnalUTXO.size());
                tx.vin[i] = additionnalUTXO[targetUTXO - allUTXO.size()];
            }
            if (fuzzed_data_provider.ConsumeBool()) {
                tx.vin[i].nSequence = fuzzed_data_provider.ConsumeIntegral<uint32_t>();
            }
            if (fuzzed_data_provider.ConsumeBool()) {
                tx.vin[i].prevout.n = fuzzed_data_provider.ConsumeIntegral<uint32_t>();
            }
            if (fuzzed_data_provider.ConsumeBool()) {
                tx.vin[i].prevout.hash = Txid::FromUint256(ConsumeUInt256(fuzzed_data_provider));
            }
            if (fuzzed_data_provider.ConsumeBool()) {
                auto scriptSig = ConsumeRandomLengthByteVector<unsigned char>(fuzzed_data_provider,
                100);
                tx.vin[i].scriptSig = CScript(scriptSig.begin(), scriptSig.end());;
            }
            if (fuzzed_data_provider.ConsumeBool()) {
                tx.vin[i].scriptWitness.stack.clear();
                int numWit = fuzzed_data_provider.ConsumeIntegralInRange<int>(0, 10);
                for (int j = 0; j < numWit; j++) {
```

```
                    tx.vin[i].scriptWitness.stack.push_back(ConsumeRandomLengthByteVector<unsigned
                    char>(fuzzed_data_provider, 100));
                }
            }
        }
    }

    int numOutput = fuzzed_data_provider.ConsumeIntegralInRange<int>(1, 10);
    tx.vout.resize(numOutput);
    for (int i = 0; i < numOutput; i++) {
        tx.vout[i].nValue = fuzzed_data_provider.ConsumeIntegral<int64_t>();

        switch(fuzzed_data_provider.ConsumeIntegralInRange<int>(0, 4)) {
            case 0:
                // P2WSH
                tx.vout[i].scriptPubKey = P2WSH_OP_TRUE;
                break;
            case 1:
                // P2SH
                tx.vout[i].scriptPubKey = P2SH_OP_TRUE;
                break;
            case 2:
                // TAPSCRIPT
                tx.vout[i].scriptPubKey = TAPROOT_OP_TRUE;
                break;
            case 3:
                // NoScript
                tx.vout[i].scriptPubKey = CScript();
                break;
            default: {
                auto scriptPubKey = ConsumeRandomLengthByteVector<unsigned char>(fuzzed_data_provider,
                100);
                tx.vout[i].scriptPubKey = CScript(scriptPubKey.begin(), scriptPubKey.end());;
                break;
            }
        }
    }

    auto res = MakeTransactionRef(tx);

    if (!coinbase) {
        // do it now, when the hash of the transaction will not change anymore
        for (int i = 0; i < numOutput; i++) {
            additionnalUTXO.emplace_back(getResolvUTXO(*res, i));
        }
    }

    return res;
}
```

H. Fuzzers Corpus Comparison ActivateBestChainStep

This annex present fuzzers individual results on the `activate_best_chain_step` harness. From a fuzzing perspective, libFuzzer again tends to "spam" the broker with many inputs that do not directly contribute to coverage. As shown in Table 35, AFL++ submits roughly four times fewer inputs but achieves twice as many accepted inputs, covering more significant code areas. As observed with the previous harness, Honggfuzz aggressively discovers new coverage.

fuzzer	inputs	#funcs	#regions	#lines	#inst	#branches	#mcdc
Initial	1/1	1302	6544	9415	1969	2279	65
Honggfuzz	281/2789	420	3827	5174	576	1470	144
AFL++	10/801	9	86	143	24	58	4
libFuzzer	22/3764	3	42	55	14	98	15
Total	**7355**	**1734**	**10499**	**14787**	**2583**	**3905**	**228**

Table 35: Inputs and coverage per Fuzzers on `activate_best_chain_step`

I. Glossary

- **TRUC**: Topologically Restricted Until Confirmation
- **PoW**: Proof of Work
- **Block in Flight**: Block announced by a peer through a CMPTBLOCK message, but which content has not yet been received the local node. Indeed, the local node have to require the content through a BLOCKTXN request. The block is considered as "in flight" until the content is received.
- **ATMP**: Accept To Memory Pool Argument, is a structure used in the validation system to accept or reject a transaction from entering the memory pool.
- **IBD**: Initial Block Download, is the process by which a new node joining the Bitcoin network downloads and verifies the entire blockchain from the genesis block to the current tip.
- **BIP**: Bitcoin Improvement Proposal, is a design document providing information to the Bitcoin community, or describing a new feature for Bitcoin.
- **BGP**: Border Gateway Protocol, is the protocol used to exchange routing information between different autonomous systems (AS) on the internet.
- **ASMAP**: Autonomous System Mapping, is a technique used to map IP addresses to their corresponding autonomous systems (AS) for network analysis and security purposes.
- **SPV**: Simplified Payment Verification, is a method that allows lightweight clients to verify transactions without downloading the entire blockchain.
- **UTXO**: Unspent Transaction Output, is a transaction output that has not been spent and can be used as an input in a new transaction.
- **ECDSA**: Elliptic Curve Digital Signature Algorithm, is a cryptographic algorithm used to generate digital signatures for verifying the authenticity and integrity of messages or transactions.
- **MCDC**: Modified Condition/Decision Coverage, is a software testing criterion that requires all possible outcomes of each condition in a decision to be tested at least once.
- **USDT**: User Defined Tracepoints, is a mechanism that allows developers to define custom tracepoints in their code for performance monitoring and debugging purposes.
- **eBPF**: Extended Berkeley Packet Filter, is a technology that allows the execution of user-defined programs in the kernel space for various purposes, including networking, security, and performance monitoring.
- **IR**: Intermediate Representation, is a data structure or code representation used in compilers and interpreters to facilitate the translation of high-level programming languages into machine code.

Bibliography

[1] B. C. Community, “Bitcoin Core Reference Implementation.” [Online]. Available: https://github.com/bitcoin/bitcoin

[2] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” [Online]. Available: http://bitcoin.org/bitcoin.pdf

[3] S. Nakamoto, “Annotated version of Satoshi's original released codebase..” [Online]. Available: https://github.com/JeremyRubin/satoshis-version

[4] “Proof of work.” [Online]. Available: https://en.wikipedia.org/wiki/Proof_of_work

[5] “OSTIF: Open Source Technology Improvement Fund,” [Online]. Available: https://ostif.org/

[6] “Brink,” [Online]. Available: https://brink.dev/

[7] “What is the Bitcoin mempool, and how does it work?.” [Online]. Available: https://cointelegraph.com/learn/articles/what-is-the-bitcoin-mempool

[8] “What is chain reorganization in blockchain technology?.” [Online]. Available: https://cointelegraph.com/explained/what-is-chain-reorganization-in-blockchain-technology

[9] B. C. Community, “BitcoinCore Security Advisories.” [Online]. Available: https://bitcoincore.org/en/security-advisories/

[10] “Delving Bitcoin.” [Online]. Available: https://delvingbitcoin.org/

[11] “libbitcoinkernel Project.” [Online]. Available: https://github.com/orgs/bitcoin/projects/3/views/1

[12] “Boost Test Library: Unit Test Framework.” [Online]. Available: https://www.boost.org/doc/libs/1_35_0/libs/test/doc/components/utf/index.html

[13] “libFuzzer - a library for coverage-guided fuzz testing.” [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[14] “AddressSanitizer (ASan).” [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[15] “UndefinedBehaviorSanitizer (ubsan).” [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[16] Google, “OSS-Fuzz: Continuous Fuzzing for Open Source Software.” [Online]. Available: https://github.com/google/oss-fuzz

[17] N. Gögge, “Btcser: A descriptor language for Bitcoin's serialization format.” [Online]. Available: https://github.com/dergoegge/btcser

[18] N. Gögge, “Semsan: Fuzz Driven Characterization Testing.” [Online]. Available: https://github.com/dergoegge/semsan

[19] N. Gögge, “Structure aware fuzzing with libprotobuf-mutator.” [Online]. Available: https://github.com/bitcoin/bitcoin/pull/26975

[20] N. Gögge, “Fuzzamoto: Holistic Fuzzing for Bitcoin Protocol Implementations..” [Online]. Available: https://github.com/dergoegge/fuzzamoto

[21] Google, “Library for structured fuzzing with protobuffers.” [Online]. Available: https://github.com/google/libprotobuf-mutator

[22] B. Garcia, “bitcoinfuzz: Differential Fuzzing of Bitcoin protocol implementations and libraries.” [Online]. Available: https://github.com/bitcoinfuzz/bitcoinfuzz

[23] “Extended Berkeley Packet Filter (eBPF).” [Online]. Available: https://en.wikipedia.org/wiki/EBPF

[24] “D-Bus.” [Online]. Available: https://en.wikipedia.org/wiki/D-Bus

[25] B. C. Community, “secp256k1.” [Online]. Available: https://github.com/bitcoin-core/secp256k1

[26] J. S. Dhruv Mehta Tim Ruffing and P. Wuille, “BIP-324: Version 2 P2P Encrypted Transport Protocol.” [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0324.mediawiki

[27] “Eclipse attacks.” [Online]. Available: https://bitcoinops.org/en/topics/eclipse-attacks/

[28] “Erebus attacks.” [Online]. Available: https://www.bitnovo.com/blog/en/what-is-an-erebus-attack

[29] “What is ASMAP in Bitcoin.” [Online]. Available: https://www.learnbitcoin.com/glossary/asmap

[30] “Package Mempool Accept Post.” [Online]. Available: https://gist.github.com/glozow/dc4e9d5c5b14ade7cdfac40f43adb18a

[31] “Transaction pinning.” [Online]. Available: https://bitcoinops.org/en/topics/transaction-pinning/

[32] “Thread Safety Analysis.” [Online]. Available: https://clang.llvm.org/docs/ThreadSafetyAnalysis.html

[33] “Static Application Security Testing.” [Online]. Available: https://semgrep.dev/solutions/static-application-security-testing/

[34] “Semgrep's Community Edition rules.” [Online]. Available: https://github.com/semgrep/semgrep-rules

[35] A. Groce, K. Jain, R. van Tonder, G. T. Kalburgi, and C. Le Goues, “Looking for lacunae in Bitcoin core's fuzzing efforts,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 185–186. doi: 10.1145/3510457.3513072.

[36] A. Crump, “SBFT'25 Fuzzing Competition: Results and Post-Mortem.” [Online]. Available: https://docs.google.com/presentation/d/1htR68976-YPnMFd_ZJeFhRewLFkKJlwprE7xwKa7A9g/edit?slide=id.p1#slide=id.p1

[37] R. David, R. A. Chaaya, and C. Heitman, “PASTIS: A Collaborative Approach to Combine Heterogeneous Software Testing Techniques,” in *IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT@ICSE 2023, Melbourne, Australia, May 14, 2023*, IEEE, May 2023, pp. 17–24. doi: 10.1109/SBFT59156.2023.00014.

[38] Y. Chen *et al.*, “EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers,” in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1967–1983. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[39] P. Goodman, G. Grieco, and A. Groce, “Tutorial: DeepState: Bringing Vulnerability Detection Tools into the Development Cycle,” in *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*, 2018, pp. 130–131. doi: 10.1109/SecDev.2018.00028.

[40] “Honggfuzz.” [Online]. Available: https://github.com/google/honggfuzz

[41] “TritonDSE: Triton-based DSE library.” [Online]. Available: https://github.com/quarkslab/tritondse

[42] “QBDI: QuarkslaB Dynamic binary Instrumentation.” [Online]. Available: https://qbdi.quarkslab.com/

[43] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, “REDQUEEN: Fuzzing with Input-to-State Correspondence,” in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[44] “Bitcoin Miniscript.” [Online]. Available: https://bitcoin.sipa.be/miniscript/

[45] “Grammar Mutator - AFL++.” [Online]. Available: https://github.com/AFLplusplus/Grammar-Mutator

[46] “NIST: Cryptographic Algorithm Validation Program.” [Online]. Available: https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program

[47] “Wycheproof - Cryptographic Test Against Known Attacks.” [Online]. Available: https://github.com/C2SP/wycheproof

[48] “Authenticated encryption with associated data (AEAD) algorithm.” [Online]. Available: https://en.wikipedia.org/wiki/ChaCha20-Poly1305

[49] “SHA-2.” [Online]. Available: https://en.wikipedia.org/wiki/SHA-2

[50] “BIP-324: Version 2 P2P Encrypted Transport Protocol.” [Online]. Available: https://bips.dev/324/

[51] “crypto-condor: a test suite for cryptographic primitives.” [Online]. Available: https://quarkslab.github.io/crypto-condor/latest/index.html

[52] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, in CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 967–980. doi: 10.1145/2508859.2516734.

[53] P. L'Ecuyer and R. Simard, "TestU01: A C library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, Aug. 2007, doi: 10.1145/1268776.1268777.

[54] "Cryptofuzz - Differential cryptography fuzzing." [Online]. Available: https://github.com/MozillaSecurity/cryptofuzz

[55] "QEMU: A generic and open source machine emulator and virtualizer." [Online]. Available: https://github.com/openssl/openssl

[56] "DeltAFLy: Differential fuzzing for cryptography." [Online]. Available: https://blog.quarkslab.com/differential-fuzzing-for-cryptography.html

[57] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1051–1065. doi: 10.1145/3548606.3560602.

[58] "Forward-secure ChaCha20 (Bitcoin's implementation)." [Online]. Available: https://github.com/bitcoin/bitcoin/blob/a33bd767a37dccf39a094d03c2f62ea81633410f/src/crypto/chacha20.h#L120

[59] "BTCD - Alternative full node bitcoin implementation written in Go (golang)." [Online]. Available: https://github.com/btcsuite/btcd

[60] "BTCD - Alternative bitcoin implementation written in Rust." [Online]. Available: https://github.com/rust-bitcoin/rust-bitcoin

[61] "Alternative bitcoin implementation written in Scala." [Online]. Available: https://bitcoin-s.org/

[62] "BTCD - Alternative bitcoin implementation written in C#." [Online]. Available: https://github.com/MetacoSA/NBitcoin

[63] "Code of the different custom implementations of cryptographic primitives in bitcoin." [Online]. Available: https://github.com/bitcoin/bitcoin/tree/master/src/crypto

[64] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 2597–2614. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[65] N. Gögge, "Fuzzamoto Grammar IR.." [Online]. Available: https://github.com/dergoegge/fuzzamoto/tree/ir/fuzzamoto-ir

[66] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, “Eclipse Attacks on Bitcoin's Peer-to-Peer Network,” in *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, 2015, pp. 129–144. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman