Looking for Lacunae in Bitcoin Core's Fuzzing Efforts

Alex Groce
Northern Arizona University
United States

Kush Jain
Carnegie Mellon University
United States

Rijnard van Tonder
Sourcegraph, Inc.
United States

Goutamkumar Tulajappa Kalburgi
Northern Arizona University
United States

Claire Le Goues
Carnegie Mellon University
United States

ABSTRACT

Bitcoin is one of the most prominent distributed software systems in the world, and a key part of a potentially revolutionary new form of financial tool, cryptocurrency. At heart, Bitcoin exists as a set of nodes running an implementation of the Bitcoin protocol. This paper describes an effort to investigate and enhance the effectiveness of the Bitcoin Core implementation fuzzing effort. The effort initially began as a query about how to escape *saturation* in the fuzzing effort, but developed into a more general exploration once it was determined that saturation was largely illusory, a byproduct of the (then) fuzzing configuration. This paper reports the process and outcomes of the two-week focused effort that emerged from that initial contact between Chaincode Labs and academic researchers. That effort found no smoking guns indicating major test/fuzz weaknesses. However, it produced a large number of additional fuzz corpus entries to add to the Bitcoin QA assets, clarified some long-standing problems in OSS-Fuzz triage, increased the set of documented fuzzers used in Bitcoin Core testing, and ran the first (to our knowledge) mutation analysis of Bitcoin Core's fuzz targets, revealing opportunities for further improvement. We contrast the Bitcoin Core transaction verification testing with that for other popular cryptocurrencies. This paper provides an overview of the challenges involved in improving testing infrastructure, processes, and documentation for a highly visible open source target system, from both the state-of-the-art research perspective and the practical engineering perspective. One major conclusion is that for well-designed fuzzing efforts, improvements to the *oracle* side of testing, increasing invariant checks and assertions, may be the best route to getting more out of fuzzing.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**.

KEYWORDS

fuzzing, saturation, test diversity, mutation analysis, oracle strength

ACM Reference Format:
Alex Groce, Kush Jain, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2021. Looking for Lacunae in Bitcoin Core's Fuzzing Efforts. In ,. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486607.3486772

1 INTRODUCTION

Bitcoin [30] is the most popular cryptocurrency, and one of the most visible (if controversial) "new" systems based on software to rise to prominence in the last decade. As we write, while volatile, Bitcoin consistently has a market cap of over half a trillion dollars since January of 2021. Due to its distributed, decentralized nature, Bitcoin in some sense is the sum of the operations of the code executed by many independent Bitcoin nodes, especially nodes that mine cryptocurrency. Bitcoin Core (https://github.com/Bitcoin/Bitcoin) is by far the most popular implementation, and serves as a reference for all other implementations. To a significant degree, the code of Bitcoin Core *is* Bitcoin. The main Bitcoin Core repo on GitHub has over 57,000 stars, and has been forked more than 30,000 times.

Because of its fame and the high monetary value of Bitcoins, the Bitcoin protocol and its implementations are a high-value target for hackers (or even nation states interested in controlling cryptocurrency developments). Therefore, testing the code is of paramount importance, including extensive functional tests and aggressive *fuzzing*. This paper describes a focused effort to identify weaknesses in, and improve, the fuzzing of Bitcoin Core.

2 INITIAL CONTACT AND THE PROBLEM OF SATURATION

Chaincode Labs (https://chaincode.com/) is a private R&D center based in Manhattan that exists solely to support and develop Bitcoin. In March of 2021, Adam Jonas, the head of special projects at Chaincode, contacted the first author to discuss determining a strategy to improve the fuzzing of Bitcoin Core. In particular, at the time, it seemed that the fuzzing was "stuck": neither code coverage nor found bugs were increasing with additional fuzzing time. After some discussion, an 80 hour effort was determined as a reasonable scope for an external, research-oriented, look at the fuzzing effort. Before that effort, conducted over the summer of 2021, began, the problem of saturation resolved itself. Nonetheless, the issue that drove the initial desire for a researcher investigation is well worth examining. Moreover, understanding why Bitcoin Core fuzzing was, temporarily but not fundamentally, saturated, may be useful to help other fuzzing campaigns avoid the same false saturation problem.

ICSE SEIP '22, Software Engineering In Practice

ACM ISBN 978-1-4503-9110-8/21/10…$15.00
https://doi.org/10.1145/3486607.3486772

Saturation, as defined in the blog post (https://blog.regehr.org/archives/1796) that brought Chaincode Labs to the first author, is when "We apply a fuzzer to some non-trivial system... initially it finds a lot of bugs... [but] the number of new bugs found by the fuzzer drops off, eventually approaching zero." That is, at first a particular fuzzer[1] applied to a system will tend to continuously, sometimes impressively, increase both coverage and discovery of previously-unknown bugs. But, at some point, these bugs are known (and often fixed) and the fuzzer stops producing new bugs. Code and behavioral coverage seems to be *saturated*.

The underlying reason for saturation is that any fuzzer (or other test generator) explores a space of generated tests according to some, perhaps very complex, probability distribution. Some bugs lie in the high-probability portion of this space, and other bugs lie in very low probability or even zero probability parts of the space. Unsurprisingly, eventually the high probability space is well-explored, and the remaining bugs are found only very infrequently, if ever. The underlying empirical facts are cruel, as noted by Böhme and Falk: "[W]ith twice the machines, we can find *all known* bugs in half the time. Yet, finding linearly *more* bugs in the same time requires exponentially more machines" [4].

Chaincode Labs saw that code coverage and bugs were not increasing in their fuzzer runs, and wanted to break out of the saturation trap. The blog post suggested several methods for doing just that, and this paper explores some of them. Note that even though the fuzzing was not saturated, the methods for escaping saturation are also useful things to try in any fuzzing effort where finding as many bugs as possible is actually important.

2.1 How Long Should You Run a Fuzzer?

One problem for inexperienced fuzzer users is that it is not always clear just how long a fuzzer needs to run. Anyone used to popular random testing tools, such as QuickCheck [9], may expect a "reasonable" budget to be on the order of hundreds of tests or at most a few minutes [24]. The fuzz testing research community has settled, to some extent, on 24 hours as a basis for evaluation of fuzzers [26], but even this may be considered a low-budget run in a serious campaign! The Solidity compiler fuzzing effort discussed in the saturation blog post (https://blog.trailofbits.com/2020/06/05/breaking-the-solidity-compiler-with-a-fuzzer/, https://blog.trailofbits.com/2021/03/23/a-year-in-the-life-of-a-compiler-fuzzing-campaign/) found many bugs only after running a specialized version of AFL for over *a month. Good fuzzing takes a lot of time.*

One thing that quickly emerged from discussions with Chaincode before the primary 80 hour effort was the limited extent of the fuzzer runs performed in early April. The fuzzing includes a large number of targets, each with its own fuzz harness and executable. At the time, the basic strategy was to run libFuzzer on each of these for 100,000 iterations. Because some targets are very fast and a few, such as full message processing, are slow, this meant in practice fuzzing most targets for only 30-90 seconds, and even the slowest targets for only a little over an hour. The total time for over 100 targets was not negligible, but expecting such short runs for each target, after an initial exploration of the easy part of the probability space, to gain coverage or bugs very often, was simply unrealistic. In particular, for complex, critical targets such as transaction verification and end-to-end message processing, 100,000 iterations was completely insufficient. When measuring tests created by humans, or even stored by a fuzzer or search-based test generation tool as interesting, 100,000 is a very large number of tests. When measuring inputs (technically individual tests of a kind) generated during fuzzing or random testing, 100,000 may be a very small number of tests.

The first suggestion for escaping coverage, therefore was very simple: run the fuzzer longer! The Chaincode team immediately tried increasing their configuration to 5 million iterations, multiplying the number of executions, and runtime, by a factor of 50. Based on initial success with a few targets, this was done for all targets, and eventually became the new default. To a large extent, saturation was no longer a problem. This exploration of simply increasing the fuzzing budget came early, about 15 days after the initial contact. The Chaincode team also added new seeds by, as advised, running more fuzzers, including AFL and Honggfuzz, in the same time frame.

By May 20th, Bitcoin Core was also in OSS-Fuzz (it was not at the time of the first discussions, due to reporting requirements, but negotiations settled this problem): https://github.com/google/oss-fuzz/tree/master/projects/bitcoin-core. From then on, Bitcoin Core has essentially been continuously fuzzed, and OSS-Fuzz quickly produced new crashes to investigate, and continues to do so: https://bugs.chromium.org/p/oss-fuzz/issues/list?q=bitcoin.

3 ADDING FUZZER DIVERSITY: USING ECLIPSER AND TRYING ENSEMBLE FUZZING

The most obvious solution when fuzzer A (e.g., libFuzzer) faces saturation on a target code-base is to bring in fuzzer B (e.g., Honggfuzz). In fact, the original blog post that attracted Chaincode's attention mentions this in the very definition of saturation: "Subsequently, a different fuzzer, applied to the same system, finds a lot of bugs." Of course, this is the easy case, and there can also be saturation across the meta-fuzzer defined as a composition of all fuzzers applied! Still, "throw in another fuzzer" is almost always a good idea when performing serious fuzzing campaigns. If that fuzzer adds some elements that are not present in any of the previously applied fuzzers, so much the better.

3.1 Fuzzing with Eclipser

Eclipser [8] is a fuzzer that combines AFL-like[2] coverage-driven mutation-based fuzzing with a scalable form of grey-box concolic testing. AFL++ [12] and libFuzzer https://llvm.org/docs/LibFuzzer.html, the two fuzzers most aggressively applied to the Bitcoin Core code, do not perform any kind of symbolic or concolic testing. Eclipser is fairly easy to apply to new systems, since it uses QEMU to run on an ordinary binary, so the first order of business was to see if it could find new bugs or at least cover code AFL++ and libFuzzer were not able to reach.

[1] The term "fuzzer" can apply to either a fuzz engine such as libFuzzer or to a "fuzz harness/target" for the fuzzed program [18]. By "fuzzer" we mean a particular *fuzz engine*, though the definition of saturation works in either case.

[2] In fact, the latest version of Eclipser simply uses AFL for non concolic fuzzing.

Either directly (from runs using Eclipser itself) or indirectly (performing long AFL or libFuzzer runs seeded with Eclipser-generated tests), Eclipser produced thousands of new corpus seed files that were accepted as PRs to merge into the Bitcoin Core QA assets repo https://github.com/bitcoin-core/qa-assets. The first author is now the third-largest contributor to the QA assets repo, in fact. These files added hundreds of new coverage edges to the basis for OSS-Fuzz and other testing of Bitcoin Core.

On the other hand, these files added *no* new covered statements; the additions were all edges, not exploration of completely uncovered code, the most promising form of new coverage [1, 15]. Nor did these tests expose any new bugs. Of course, in the long run testing in OSS-Fuzz or on the QA team's servers may use these tests to help cover new code or find new bugs, but the immediate impact is more modest.

One of the outcomes of the 80 hour effort was full documentation of how to run Eclipser v1.x on the Bitcoin Core code: https://github.com/bitcoin/bitcoin/blob/master/doc/fuzzing.md#fuzzing-bitcoin-core-using-eclipser-v1x.

3.2 Ensemble Fuzzing

Given that Eclipser, while useful, did not produce dramatic results, simply running and adding instructions for more fuzzers did not seem like a particularly productive use of much more of the two-week effort. However, because Eclipser was somewhat useful, one obvious way to exploit multiple fuzzers was to try *ensemble fuzzing*.

Ensemble fuzzing [7] is an approach that recognizes the need for diverse methods for test generation, at least in the context of fuzzing; using multiple fuzzers to seed each other and avoid saturation is a core motivation for ensemble fuzzing. Inspired by ensemble methods in machine learning [10], ensemble fuzzing runs multiple fuzzers, and uses inputs generated by each fuzzer to seed the other fuzzers. Ensemble fuzzing is currently (in principle) supported by the Enfuzz website (http://wingtecher.com/Enfuzz) and by the DeepState [14] front-end.[3]

However, the Enfuzz website has never, since the Bitcoin Core effort began, actually been up and working, and various build and library version errors made it impossible to get the GitHub version (https://github.com/enfuzz/enfuzz) to work, either. DeepState's ensemble code is more limited, but works. However, it would require re-writing Bitcoin Core test harnesses to use DeepState's GoogleTest-like API. In theory, this is not a huge burden: the custom fuzzing core API built by the Bitcoin Core team is fairly similar, in concept, to DeepState, providing ways to obtain values of various types using different fuzzer back-ends. However, there are numerous small differences and specialized behaviors in the Bitcoin code that would require substantial work to re-engineer, and DeepState would add a significant build dependency to the Bitcoin testing infrastructure; such a change might well not make it through the Bitcoin Core approval process, once done. Further, the effort to re-write more than 100 targets would be substantial, even if most targets were fairly trivial to translate. Building a one-off version not intended to be rolled into the main code-base might be worthwhile, but would clearly take more than the 80 hours available, for unknown payoff. Therefore performing ensemble fuzzing of Bitcoin Core remains future work for the QA team, or awaits the arrival of more reliable ensemble fuzzers that operate through a libFuzzer, AFL, or Honggfuzz interface that requires no changes on the Bitcoin Core side.

Note that in one important sense Bitcoin Core *is* using ensemble fuzzing, in that OSS-Fuzz runs multiple fuzzers, including libFuzzer, AFL, and Honggfuzz) with different compilation flags and sanitizers. Additionally, the Bitcoin Core team has servers running different fuzzers. All of these are coordinated via the `qa-assets` repository to which our Eclipser-based tests were added. This is, however, a more manual and less controlled process than true ensemble cross-seeding on-the-fly during a fuzzing campaign, and there are suggestions that a well-chosen coordination strategy can significantly improve ensemble effectiveness.

4 TRYING SWARM FUZZING

Swarm testing [23] is a method for improving test generation that relies on identifying *features* [22] of tests, and disabling some of the features in each test. For instance, if features are API calls, and we are testing a stack with `push`, `pop`, `top`, and `clear` calls, a non-swarm random test of any significant length will contain multiple calls to *all* of the functions. In swarm testing, however, for each test *some of the calls (with probability usually equal to 0.5 for each call) will be disabled*, but *different calls* will be disabled for *each generated test*. This produces less variance between calls *within tests*, but much more variance *between tests*. Practically, in the stack example, it will enable the size of the stack to grow much larger than it ever would have any chance of doing in non-swarm testing, due to some tests omitting `pop` and/or `clear` calls. Swarm testing is widely used in compiler testing [27] and is a core element of the testing for FoundationDB [32].

Some Bitcoin Core fuzz harnesses resemble API-sequence generation. In particular, the `process_messages` harness produces method sequences of a fixed length, where each message is of a particular type (there are 34 distinct message types, again similar to an API-call "fuzz surface"; see Figure 1). Swarm testing is an obvious candidate approach for enhancing this important fuzz harness. Inspection of the `process_messages` code reveals that not only does the harness (https://github.com/bitcoin/bitcoin/blob/master/src/test/fuzz/process_messages.cpp) not perform swarm testing, it doesn't even pick from the message types; it simply generates an arbitrary string that will be parsed as a message type, a fuzzer-controlled arbitrary number of times:

```
while (fuzzed_data_provider.ConsumeBool()) {
  const std::string random_message_type{
    fuzzed_data_provider.ConsumeBytesAsString(
      CMessageHeader::COMMAND_SIZE).c_str()};
```

DeepState [14] (https://github.com/trailofbits/deepstate) provides strong support for swarm fuzzing. In DeepState, transforming the call to `ConsumeBytesAsString` to a `OneOf` nondeterministic choice operator over the strings defined in Figure 1, and compiling the harness with `-DDEEPSTATE_PURE_SWARM` would enable automatic fuzzer controlled swarm testing. However, as with ensemble fuzzing, while there are similarities to the Bitcoin Core infrastructure for fuzz targets and the DeepState API, re-writing the fuzz

[3] See https://blog.trailofbits.com/2019/09/03/deepstate-now-supports-ensemble-fuzzing/.

```
namespace NetMsgType {
  extern const char* VERSION;
  extern const char* VERACK;
  extern const char* ADDR;
  extern const char *ADDRV2;
  extern const char *SENDADDRV2;
  extern const char* INV;
  extern const char* GETDATA;
  extern const char* MERKLEBLOCK;
  extern const char* GETBLOCKS;
  extern const char* GETHEADERS;
  extern const char* TX;
  extern const char* HEADERS;
  extern const char* BLOCK;
  extern const char* GETADDR;
  extern const char* MEMPOOL;
  extern const char* PING;
  extern const char* PONG;
  extern const char* NOTFOUND;
  extern const char* FILTERLOAD;
  extern const char* FILTERADD;
  extern const char* FILTERCLEAR;
  extern const char* SENDHEADERS;
  extern const char* FEEFILTER;
  extern const char* SENDCMPCT;
  extern const char* CMPCTBLOCK;
  extern const char* GETBLOCKTXN;
  extern const char* BLOCKTXN;
  extern const char* GETCFILTERS;
  extern const char* CFILTER;
  extern const char* GETCFHEADERS;
  extern const char* CFHEADERS;
  extern const char* GETCFCHECKPT;
  extern const char* CFCHECKPT;
  extern const char* WTXIDRELAY;
  }
```

Figure 1: The message types declaration in `src/protocol.h`.

targets to use DeepState seemed, and change many string generation calls to `OneOf` selections was too large a time investment for the likely payoff without first trying a less ambitious approach.

The solution was to focus on developing a swarm harness for the most promising target, `process_messages`, alone, and determine if an aggressive campaign on that target would produce substantial new coverage, or even unknown bugs. This `process_messages_swarm` harness required modest effort, and could show if the strategy was worth devoting more resources to applying more broadly. After four weeks of fuzzing, however, the new coverage generated was minimal; less than would be expected from simply devoting that level of effort to the original `process_messages` target. Why?

First, it is likely that the automatic translation of more than 10,000 corpus files for `process_messages` was imperfect. Parsing raw bit-streams to find commands is hard, and writing a dynamic parser to use in-flight data to extract the meaning of each test was not feasible in the 80 hour effort. This probably meant some corpus files lost some or all of their value, leaving the new fuzz effort behind the equivalent `process_messages` fuzz.

Most importantly, however, the `process_messages` corpus is regularly *cross-seeded* with data from specialized fuzz targets that generate only one type of message, e.g., such as the tests in https://github.com/bitcoin-core/qa-assets/tree/main/fuzz_seed_corpus/process_message_mempool, which correspond to `extern const char* MEMPOOL` in Figure 1 . This not only provides some of the power of swarm testing, it achieves a second goal of the swarm harness. Namely, one concern was that since `process_message` and `process_messages` generate raw strings that may not even have a valid message type, the fuzzers spend too much effort fuzzing the type to little benefit. The specialized targets avoid this problem by constraining the fuzzing to only generate one type of message when running in the specialized mode. The ability to run `process_message` with a single message type and the frequent introduction of the resulting inputs into `process_messages` (and the generic, unconstrained `process_message` fuzzing) probably, in a less automated way, also achieves many of the benefits of swarm testing itself: mixing complex lengthy runs of a single type or mix of types. Where introducing "real" swarm testing is difficult, but fuzzing infrastructure supports this mode of operation, it may be a useful alternative. It is on the other hand unclear how many fuzz frameworks are as sophisticated as Bitcoin Core's, making this possible; in many cases simply implementing swarm directly (or writing the harnesses using DeepState) would likely be easier. It does show that some advanced fuzzing strategies can be anticipated by particularly savvy and capable fuzz engineers, willing to directly use raw fuzzing data, and write tools to support that kind of low-level hand-tuning of fuzz corpuses.

5 SIDE ISSUES: SPURIOUS BUGS, FUZZER MYSTERIES, AND AFL STABILITY

A critical point about testing is that it is like other kinds of software development. In practice, while adding "features" or at least optimizations of existing features — in our context, adding new fuzzers or cutting-edge methods such as swarm testing — is usually seen as the most interesting aspect of the work, the reality is that developer/tester time is often spent on more mundane, frustrating problems — e.g., working around bugs in test infrastructure itself.

For example, in the Bitcoin Core effort, early attempts to apply Eclipser and AFL ran into a problem: at some point both fuzzers would essentially stop working and begin producing a vast number of spurious crashes. By spurious crashes, we mean un-reproducible crashes: inputs that the fuzzer marks as causing a crash, but which, when executed in a non-fuzzing context using the fuzz harness, do not crash. Spurious crashes are not unlike the phenomenon of *flaky tests* [11], one of the banes of automated software testing. A flaky test is a test that, for the same snapshot of test code and code under test, sometimes fails and sometimes passes. In fact, arguably, spurious crashes in fuzzing are *flaky tests* where the flakiness is due to an environmental dependence: the test fails when run under the fuzzer. Flaky tests and spurious crashes consume significant testing and developer/test engineer resources. During the very first discussions with the Chaincode team, when Chaincode first ran AFL, some spurious crashes were produced; these were eventually just discarded without understanding the root cause. This happens to many spurious crashes, unlike flaky tests which manifest in standing regression tests that often have to be either removed or otherwise mitigated. In fuzzing, simply throwing away spurious crashes without understanding them is a more feasible option, though one that may hide some detected vulnerabilities. OSS-Fuzz runs using AFL, also produced some spurious crashes.

Unlike most spurious crashes, these were eventually (partly) understood. Because AFL and Eclipser, run more than a few hours, inevitably reached as stage where they *only* produced spurious crashes (instead of doing any useful fuzzing), the problem had to be understood, if these fuzzers were to be useful. The first author discovered that the AFL and Eclipser problems were, strictly speaking, not flaky or spurious at all. The fuzz harnesses all fail if there is insufficient storage space on `tmp` to execute the Bitcoin software. At some point, this happens, even if fuzzing begins with a large amount of free storage. The problem is that some (but not all) fuzz executions under AFL and Eclipser (but not under libFuzzer) fail to clean up /tmp/test_common_Bitcoin Core; each run that fails to clean up leaves an 18-19 megabyte footprint. Over the millions of executions involved in fuzzing, this inevitably fills available space.

This problem was also (probably) causing the original AFL issues seen by Chaincode, and (almost certainly, on examination of the OSS-Fuzz logs) the OSS-Fuzz failures that did not reproduce. Multiple possible solutions were discussed (at length) and proposed in a PR (https://github.com/bitcoin/bitcoin/pull/22472), but as we write none of these have been deemed successful. Also, the underlying reason why AFL and Eclipser sometimes fail to clean up `/tmp` is not understood; removing all threads did not fix the issue, though it might be useful for other reasons. The presence of threads may explain the relatively low stability of AFL fuzzing on Bitcoin Core, another issue raised during the 80 hour effort: https://github.com/bitcoin/bitcoin/issues/22551. Low stability indicates that when an input is executed multiple times, it does not always take the same path. The worse the stability, the less useful signal AFL is receiving from path coverage during fuzzing. Bitcoin Core's stability hovers around 80%, possibly as a result of the presence of threads in the code. Even if the threads do not cause nondeterministic behavior of the Bitcoin Core system, from a functional point of view, they may cause path coverage to vary more than AFL "likes." It is hard to guess if this has a serious adverse effect on fuzzing performance.

One reason these issues have not been resolved is that they do not affect the most effective (thus far) fuzzing approach. Most new coverage obtained on Bitcoin Core, and most bugs found, can be credited to libFuzzer, which is not affected by the `tmp` problem, and at least fails to complain about stability, even if it is affected. At present, users of Eclipser on Bitcoin Core are advised to perhaps apply one of the patches proposed in the open PR on the problem, or to manually clean `/tmp` in some other way.

6 MUTATION ANALYSIS

Attempting to improve a fuzzing effort is one way to find problems with the effort; if you succeed, you found a weakness. However, none of the attempts described above exposed a serious problem. Adding more fuzzers would be *good*, but was not obviously *essential*. True ensemble fuzzing was not feasible, and swarm testing was, for practical purposes, already performed by alternative means. An alternative is to directly look for holes in testing. The Bitcoin Core fuzzing team clearly was measuring and inspecting code coverage (see https://marcofalke.github.io/btc_cov/), so little value would be added by inspecting traditional coverage alone. Mutation testing/-analysis [5, 28, 31], however, subsumes code coverage and adds extremely valuable information on *oracle power* in addition to mere coverage [17]. This is perhaps especially valuable in fuzzing, where "you only see crashes" is a persistent concern. In previous work, we had used mutation testing to improve the random testing of the Linux kernel's RCU module, and in the process discovered some subtle kernel bugs [2, 16].

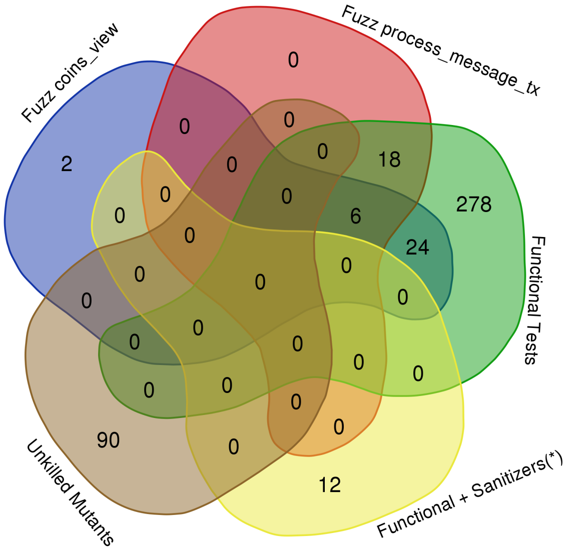


Figure 2: A comparison of mutation kills for `tx_verify.cpp` when subjected to different testing and fuzzing parameters. At least 29 of the 90 surviving mutants are equivalent, and thus not indicative of testing weaknesses. * indicates only mutants not killed by other methods were analyzed;

We used the universal mutator (https://github.com/agroce/universalmutator) [20], a mutation tool already used widely in the blockchain and smart contract world, to mutate the Bitcoin Core transaction verification code, and, subsequently, that of other popular cryptocurrencies. We focused on transaction verification/validation code, as it is generally well-covered by tests, and obviously an extremely critical functionality for any blockchain.

6.1 Mutation Testing Bitcoin

To perform mutation analysis on Bitcoin, we generated mutations for code in the `tx_verify.cpp` file. Fuzzing covers 96 of 98 lines of code, 8 of 8 functions, and 74 of 102 branches for this file, guaranteeing that mutation testing will not primarily reflect missing coverage. Comparing coverage to that for functional testing, the fuzz testing has very slightly lower branch coverage, but the numbers are almost identical (72.5% vs. 73%), and the fuzz testing covers *different* branches than the functional testing. The missing lines are different for functional and fuzz testing, as well, in fact. And, as noted above, there can be no doubt that transaction verification is a critical Bitcoin function: in fact, arguably, checking transactions for correctness is *the raison d'être* of any blockchain, and getting transactions that should be invalid past such checks would be an obvious potential attacker goal. We evaluated the ability of fuzzing and functional testing to detect mutants of `tx_verify.cpp`. Figure 2 summarizes our mutation analysis of the file.

Original mutated code (and immediate context):

```
            if (coin.IsCoinBase() && nSpendHeight - coin.nHeight < COINBASE_MATURITY) {
                    return state.Invalid(TxValidationResult::TX_PREMATURE_SPEND, "bad-txns-premature-spend-of-coinbase",
                        strprintf("tried to spend coinbase at depth %d", nSpendHeight - coin.nHeight));
```

Mutant #379 changes the `strprintf` to:

```
        strprintf("tried to spend coinbase at depth %d", nSpendHeight / coin.nHeight));
```

Mutant #380 changes the `strprintf` to:

```
        strprintf("tried to spend coinbase at depth %d", nSpendHeight % coin.nHeight));
```

Figure 3: Mutants detected *only* by fuzzing.

The universal mutator generated 430 compiling mutants of the file in less than three hours.[4]. Two fuzz targets seemed to be relevant to fuzzing `tx_verify.cpp` code: `process_message_tx` and `coins_view`. In theory `process_message` and `process_messages` were also potential interest, but the relevant corpus entries were duplicated in `process_message_tx` (we verified the two more general targets provided *no* additional mutant kills). We ran fuzzing for five minutes using libFuzzer exploration based on the full (and quite large: 4,517 tests for `process_message_tx`, and 6,889 for `coins_view`) QA asset corpus for each harness, with all sanitizers enabled. The `process_message_tx` target was able to detect 24 mutants, and the `coins_view` harness was able to detect 32 mutants, for a total of 50 mutants (since some mutants were detected by both). In other words, fuzzing could detect just under 12% of all the generated mutants. This is not necessarily a bad result: fuzzing inherently has trouble detecting subtle, non-crash-inducing, bugs in code, because writing a strong specification of correct behavior that covers all the bizarre and pointless inputs produced in fuzzing is often impractical, or would require a specification nearly as complex as the code itself. This is one reason *differential* fuzzing is promising: a reference implementation is such a specification. Bitcoin Core's cryptographic elements are, in fact, differentially fuzzed https://github.com/bitcoin/bitcoin/pull/22704#issuecomment-898989809.

A major purpose of fuzzing is, then, to address limits in more traditional functional testing, where known inputs are paired with expected behavior. While functional or unit testing is very powerful, the kinds of bugs found in vulnerabilities often involve the kind of inputs that don't appear in "normal" unit/functional tests, as shown by the success of fuzzing and security audits [19]. The real question, then, is how many mutants that survive Bitcoin Core's extensive functional tests survive fuzzing.

The answer is: not too many. The functional tests without sanitizers enabled catch an additional 278 mutants. Turning on sanitizers catches an additional 12 mutants. Only 90 of the 430 compiling mutants survive all tests, for an overall mutation score of 79.07%.

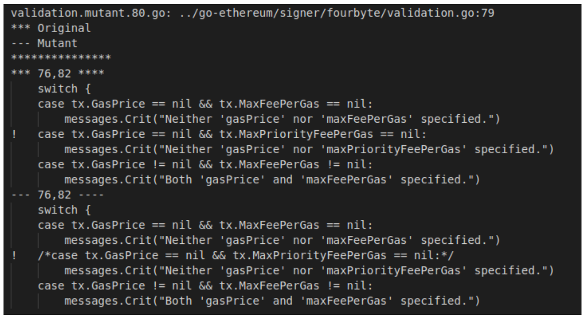
Fuzzing adds two unique mutant kills beyond those produced by the functional testing. Figure 3 shows the (very similar code) for these two mutants. Only fuzzing generates inputs that cause `coin.nHeight` to be zero. Fuzzing doesn't increase code coverage here, but does increase interesting *data value* coverage. Use of the libFuzzer `-use_value_profile=1` flag is likely instrumental in achieving such good value coverage. Adding that flag was one of the first suggestions in the 80 hour effort, but it turned out this was already standard practice in the Bitcoin Core fuzz configuration. Note that the opportunity for either approach to generate a zero value here is limited: functional tests only cover the mutated line 14 times, and fuzzing 556 times. In contrast, both cover numerous other lines of code more than a million times. For functional tests, the line is the 2nd least-covered line executed in `tx_verify.cpp`, and it is the 4th least-covered line executed for fuzzing.

Manual inspection of the 90 surviving mutants showed that at least 29 of these were clearly semantically equivalent to the un-mutated code. For instance, many mutants removed or weakened an assertion; clearly this cannot ever be detected, since it can only transform failing tests into passing tests. The full, detailed list of surviving, not-obviously-equivalent mutants, prioritized by an FPF ranking [6, 13], is available here: https://github.com/agroce/bitcorpus/blob/master/mutation/prioritized_full_inspect.txt. Discussion with the Bitcoin Core team is ongoing as we write (https://github.com/bitcoin/bitcoin/issues/22690), but thus far none of these mutants seem to expose serious testing problems. After manual pruning, the mutation score is 85.8%, and some of the remaining 61 mutants are likely also equivalent. The limitations of the fuzzing oracle are clear: fuzzing covers 98% of statements and 72.5% of branches as we write, but can kill fewer than 12% of mutants. The much greater killing power of the functional tests obviously does not lie in the marginal 0.5 percentage points of branch coverage it obtains; it lies in the ability to reject incorrect executions that do not crash or set off a sanitizer alarm.

This raises the question: why fuzz? The coverage for high quality (if imperfect) functional tests such as those for Bitcoin Core will often be considerably higher, and the oracle will almost always be *much* more powerful. The answer lies in the fact that, even in the presence of such high quality tests, fuzzing uncovers subtle bugs that functional tests designed by humans will almost never detect, e.g. https://github.com/bitcoin/bitcoin/issues/22450.[5] In part this is due to the fact that when functional tests and fuzzing have similar coverage, they often cover *different* hard-to-reach code, as in the case of `tx_verify.cpp`. Fuzzing is *not* a replacement for functional/unit tests; and functional/unit tests are not a replacement for fuzzing. In our mutation analysis, consider the two mutants detected by `coins_view` fuzzing alone. In the traditional, score-based, view of mutation analysis, the `coins_view` fuzz harness would be seen as performing badly. But it detects two (hypothetical) bugs not detectable by other means; in the real world, if one such bug is exploitable, detecting it may "pay for" all the fuzzing effort, and there will seldom be just one such bug (see https://github.com/bitcoin/bitcoin/issues?q=is%3Aissue+fuzz+is%3Aclosed+label%3ABug for an approximate list of fuzzer-detected, fixed bugs in Bitcoin Core).

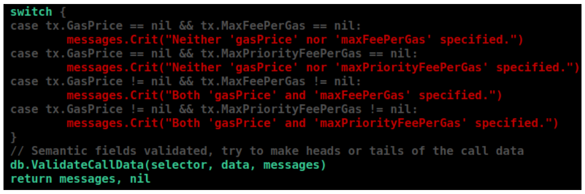
[4] Building all test and fuzz targets for each candidate mutant takes some time, and parallelizing the task would require multiple sandboxed copies of the code. Constructing a more focused build command might speed this up substantially.

[5] Comments on this bug, such as "Another win for fuzzing, oh wow." and "Fuzzer rulez!" show that the Bitcoin Core team has little doubt about the power of fuzzing.



```
validation.mutant.80.go: ../go-ethereum/signer/fourbyte/validation.go:79
*** Original
--- Mutant
***************
*** 76,82 ****
    switch {
    case tx.GasPrice == nil && tx.MaxFeePerGas == nil:
        messages.Crit("Neither 'gasPrice' nor 'maxFeePerGas' specified.")
!   case tx.GasPrice == nil && tx.MaxPriorityFeePerGas == nil:
        messages.Crit("Neither 'gasPrice' nor 'maxPriorityFeePerGas' specified.")
    case tx.GasPrice != nil && tx.MaxFeePerGas != nil:
        messages.Crit("Both 'gasPrice' and 'maxFeePerGas' specified.")
--- 76,82 ----
    switch {
    case tx.GasPrice == nil && tx.MaxFeePerGas == nil:
        messages.Crit("Neither 'gasPrice' nor 'maxFeePerGas' specified.")
!   /*case tx.GasPrice == nil && tx.MaxPriorityFeePerGas == nil:*/
        messages.Crit("Neither 'gasPrice' nor 'maxPriorityFeePerGas' specified.")
    case tx.GasPrice != nil && tx.MaxFeePerGas != nil:
        messages.Crit("Both 'gasPrice' and 'maxFeePerGas' specified.")
```

Figure 4: A mutation for Ethereum transaction validation.



```
switch {
case tx.GasPrice == nil && tx.MaxFeePerGas == nil:
        messages.Crit("Neither 'gasPrice' nor 'maxFeePerGas' specified.")
case tx.GasPrice == nil && tx.MaxPriorityFeePerGas == nil:
        messages.Crit("Neither 'gasPrice' nor 'maxPriorityFeePerGas' specified.")
case tx.GasPrice != nil && tx.MaxFeePerGas != nil:
        messages.Crit("Both 'gasPrice' and 'maxFeePerGas' specified.")
case tx.GasPrice != nil && tx.MaxPriorityFeePerGas != nil:
        messages.Crit("Both 'gasPrice' and 'maxPriorityFeePerGas' specified.")
}
// Semantic fields validated, try to make heads or tails of the call data
db.ValidateCallData(selector, data, messages)
return messages, nil
```

Figure 5: Missed coverage for mutations in Figure 4.

If the "good guys" don't fuzz well, you can be sure the bad guys will, for software protecting billions of dollars of assets.

6.2 Other Cryptocurrency Projects

To put our work on Bitcoin in context, we performed mutation analysis of transaction-verification-related code for other popular cryptocurrencies. We generated mutants using the universal mutator for code in Ethereum, Dogecoin, Avalanche, Stellar, and Cosmos implementations, a selection from the top 30 cryptocurrencies[6] Our selection was also influenced by the availability of code coverage, and indicative of the opportunities for mutation testing of Bitcoin and popular altcoins, rather than a comprehensive survey. Because it requires significant effort to set up or otherwise understand the extent of fuzz testing in this heterogeneous selection of projects, our mutation analysis here only considers mutation testing against the project test suite, not fuzzing. As discussed above, without heroic effort, fuzzing is likely to add only modest additional mutant kills.

We identified candidate files that might be roughly comparable to Bitcoin's `tx_verify.cpp` by searching for keywords like `transaction`, `verify`, `sign`, and `validate`. We manually inspected functions and test coverage for these functions (where applicable) to identify which files would be interesting targets for mutation. Ultimately we settled on one to three files per project that are representative of some interesting and tested functionality (a choice that we readily acknowledge is by no means comprehensive or suggestive of a project's quality and testing as a whole).

Setup. We ran the universal mutator on the candidate files above, using both universal and language specific rules. There were 92 universal rules (transformations that can apply to any language) and between 0 and 20 language specific rules, depending on the project source language.[7] Across all projects, our rules generated between 492 and 8,567 mutants per file. We ran mutants against each project's default test suite (determined by consulting READMEs and build/CI documentation) using universal mutator to obtain a final mutation score. The runtime of this step naturally varies depending on test suite execution speed and candidates per file, ranging from 42 minutes (go-ethereum's `validation.go`) to 32 hours (dogecoin's `bitcoin-tx.cpp`).

Results. Table 1 summarizes our results, and lists the **Mutation score**, **File coverage**, and **Project coverage**. **Mutation score** represents the proportion of generated mutants that were killed by the project's test suite divided by the total number of mutants (higher is better). File and project metrics report statement level coverage for each chosen file and the entire project. In principle, we expect that higher file coverage (i.e., more tested code) should correlate with a higher mutation score. Our investigation reveals interesting challenges in the interplay of test code, code-under-test, and reported coverage. One illustrative project (not included in our table) is Solana,[8] a codebase predominantly written in Rust. The project reports high test coverage overall (82%) and in specific files (`sigverify.rs`) with seemingly important signature verification code (74%).[9] When we applied mutation testing ti this file, we realized that both test code and code-under-test live in the same file, and further, that reported code coverage includes test code. With inline tests, we face a dual challenge of restricting mutations to only tested code (a problem best addressed by the mutation tool) and obtaining coverage restricted only to tested code (best solved by the coverage tool). These concerns naturally bear on both mutation score and code coverage, and our experience reveals the need to tease such concerns apart to accurately assess true test quality.

For illustration, Table 2 shows some of the mutation rules with examples of surviving mutants. One commonly used operator, statement deletion, comments out a line of source code. Other operators included adding `break` and `continue` to statements, along with changing binary operators such as < to ==. Investigating files with lower coverage (e.g., those in Ethereum), we noticed that many of the generated mutants removed lines related to error checking (e.g., in switch statements) that were apparently not covered by tests (cf. Figures 4 and 5).

The Bitcoin Core mutation score (without pruning or fuzzer-killed mutants) ranks high: 2nd out of 6 projects. Bitcoin Core also has the highest File and Project coverage of any project. Our experience working with Bitcoin Core developers suggests that they are pro-active about code quality and testing, which is likely to lead (directly or indirectly) to test suite quality. At the same time, code for transaction verification and validation may naturally vary depending on context, code organization, and language, as reflected by the differences in lines of code in our selection. Our mutation testing ultimately is best understood as a set of individual data points that are difficult to compare fully quantitatively across projects. We discuss these considerations in more detail next.

Discussion. For the alternative cryptocurrencies, we limited our attention to the default (typically functional) tests provided in the

[6] We chose these by market capitalization according to https://coinmarketcap.com.

[7] See https://github.com/agroce/universalmutator/tree/master/universalmutator/static

[8] https://github.com/solana-labs/solana, a top 10 cryptocurrency at the time of writing.

[9] https://app.codecov.io/gh/solana-labs/solana

Project	File path	LOC	Mutation score	File coverage	Project coverage
bitcoin	src/consensus/tx_verify.cpp	210	78.6%	98.7%	84.2%
go-ethereum	core/block_validator.go	129	70.1%	81.0%	58.8%
	signer/fourbyte/validation.go	127	49.5%	60.0%	
	signer/core/signed_data.go	1,044	25.3%	69.3%	
dogecoin	src/bitcoin-tx.cpp	847	58.7%	-†	70.1%
avalanchego	vms/platformvm/add_subnet_validator_tx.go	308	57.3%	81.0%	63.6%
stellar	src/historywork/VerifyTxResultsWork.cpp	192	85.1%	85.3%	74.9%
cosmos-sdk	x/auth/ante/sigverify.go	510	67.3%	67.0%	60.9%

Table 1: Code Coverage and Mutation Scores Across Popular Cryptocurrencies. Mutation score represents the proportion of mutants that were killed divided by the total number of mutants (higher is better). File coverage and Project coverage report statement level coverage for the file and entire project, respectively. LOC represents the lines of code of the chosen file. †We were unable to obtain individual coverage for one file.

<table>
<tr><th>Mutation Operator</th><th>Project and File</th><th>Mutation Example</th></tr>
<tr><td>Comment out source code line</td><td>bitcoin - tx_verify.cpp</td><td>if (!MoneyRange(coin.out.nValue) || !MoneyRange(nValueIn)) {
-   return state.Invalid(TxValidationResult::TX_CONSENSUS ,
-       "bad-txns-inputvalues-outofrange");
+   /* return state.Invalid(TxValidationResult::TX_CONSENSUS ,
+   "bad-txns-inputvalues-outofrange"); */
}</td></tr>
<tr><td>Replace < with ==</td><td>go-ethereum - block_validator.go</td><td>- if desiredLimit < params.MinGasLimit {
+ if desiredLimit == params.MinGasLimit {
  desiredLimit = params.MinGasLimit
}</td></tr>
<tr><td>Add continue to statement</td><td>dogecoin - bitcoin-tx.cpp</td><td>while ((!feof(f)) && (!ferror(f))) {
  char buf[4096];
+ continue;
  ...
}</td></tr>
<tr><td>Add break to statement</td><td>bitcoin - tx_verify.cpp</td><td>for (const auto& txin : tx.vin) {
+ break;
  nSigOps += txin.scriptSig.GetSigOpCount(false);
}</td></tr>
<tr><td>Flip arguments of function</td><td>go-ethereum - validation.go</td><td>if tx.Data != nil && tx.Input != nil
-       && !bytes.Equal(*tx.Data, *tx.Input) {
+   && !bytes.Equal(*tx.Input, *tx.Data) {
  return nil, errors.New(`...`)
}</td></tr>
</table>

Table 2: Sample of Mutation Rules and Examples for Various Cryptocurrencies that were not killed.

build chain, according to READMEs and other documentation. We know that at least some of the considered alternative cryptocurrencies (e.g., Ethereum) make use of fuzz testing (i.e., are enrolled in OSS-Fuzz). However, if the marginal contribution of fuzzing is similar to Bitcoin, it may be indicative of approximate mutation scores over the kinds of tests considered. At minimum, we observe that Bitcoin is likely in the upper quartile of cryptocurrencies in terms of test quality as measured by coverage and mutation score. In any event, our investigation suggests there is clearly room to improve test coverage (and mutation score) across popular cryptocurrencies.

Perhaps a more important lesson from this qualitative comparison is that mutation testing is an important way to evaluate test suite quality, especially given the complementary role of fuzz and functional testing on Bitcoin. We argue that this sort of evaluation should be a first class concern for projects like cryptocurrencies, targeted at least at central logic (like transaction validation). It also *should* be reasonably feasible, in principle; we found the problem tractable, once we isolated key functionality and a test suite. However, our experience suggests key areas that make it difficult to conduct such evaluations comprehensively: (1) identifying critical blockchain functionality and evaluating associated test coverage (2) understanding why gaps in coverage exist (is it a lack of tests, or does some other kind of testing take place (e.g., integration testing) where coverage is not recorded? Even for Bitcoin Core's relatively well-documented tests, the first author initially thought the weak unit tests. not the extensive functional tests, were the primary tests for the code. In brief: with cryptocurrency projects seeking to unabashedly upend the status quo of financial systems, it seems only reasonable that they embrace very high standards of testing; in this context, our view is that mutation testing is an effective, but easily (and evidently) overlooked vector for improving test suites.

7 CONCLUSION: HOW GOOD IS THE BITCOIN CORE TESTING AND FUZZING?

In a sense this is a fundamentally hard question to answer (software engineering research has been trying to understand how to answer this question for decades, after all): certainly, an 80 hour effort by one fuzzing researcher, who was generally familiar with blockchain and Bitcoin technology, but not at all familiar with the Bitcoin Core implementation, in terms of high level approach, testing infrastructure, or (of course) implementation details, cannot provide anything like a definitive answer to that question.

The previous section discusses our limited effort to place the Bitcoin Core test effort in the context of other cryptocurrencies, but the limits of that evaluation make it hard to draw strong conclusions. The complexities of the test efforts and the various ways the systems could be compared make this a weak basis for evaluating Bitcoin Core itself. Bitcoin Core has about 10,000 lines of fuzz harness code; Go-Ethereum has about 74KLOC of fuzz harness code; is the fuzzing thus likely to be 7 times better? We don't claim to know. Instead, we base our evaluation on the effort to improve Bitcoin Core fuzzing, itself, plus the fact that Bitcoin Core's coverage and mutation results seem very likely to be at minimum competitive with those of other high-profile cryptocurrencies.

The initial fuzzing effort, at the time of contact, was certainly operating in a suboptimal way, and hence *seemed* to be facing a difficult saturation problem. However, the simplest of industrial-fuzzing-savvy advice (run the fuzzers longer, add the system to OSS-Fuzz, try more fuzzers) was sufficient to make the fundamentally sound fuzzing basis more useful, and the advice was solicited and taken, very quickly.

Additional efforts to improve the fuzzing added some value in terms of new corpus entries, and paved the way for better documented, and more systematic, application of a variety of fuzzers. Fundamentally, however, these improvements were relatively small compared to the overall scope of Bitcoin Core testing. This was substantially due to the limited time-frame, but not due to either a lack of expertise or serious attempts to improve fuzzing, given that time-frame, we believe.

The basic cause of the limited improvement was that, at present, Bitcoin Core has a well-designed and effective fuzzing infrastructure, supported by intelligent manual augmentation of the more automated aspects of the fuzzing, and in the context of an aggressive set of functional tests with generally good code coverage.

7.1 The Bitcoin Core Fuzzing Infrastructure

The basic fuzzing infrastructure for Bitcoin resides in `src/tests/fuzz`, which consists of about 200 fuzz target implementations, ranging in size from a few lines to about 500 lines, plus common infrastructure, primarily found in `FuzzedDataProvider.h`. The fuzzing implementation is compact, but supports a common way to write fuzz targets that perform fuzz generation of C++ generic and Bitcoin-specific data structures. The generators are designed to allow different fuzzers to serve as the "back end" of the system, as in DeepState. This approach allows some targets to be very compact, e.g., see Figure 6.

```
    FUZZ_TARGET(parse_script)
{
    const std::string script_string(buffer.begin(), buffer.end());
    try
        (void)ParseScript(script_string);
     catch (const std::runtime_error&)

 }
```

Figure 6: Fuzz target for script parsing.

In two weeks, it is impossible to examine every fuzz target in detail, much less understand how they interact with the targeted code, but most inspected targets are well-designed, and in cases where targets appear to be too generic, wasting time, as with `process_message`, there seems to be manual seeding that avoids the problem of forcing fuzzers to generate too many magic strings.

One sign that the fuzzing is of reasonable quality is that the gap in code coverage between fuzzing and high-quality functional tests is not very large. For both fuzzing and functional tests, some missing coverage and some unkilled mutants are almost certainly (and stated to be such by Bitcoin Core team members to us) redundant or defensive code to protect against "impossible" problems. This is unsurprising in code intended to be highly reliable; coverage of core file system functionality was only 89% during a year of extensive random testing of a file system developed for the Curiosity Mars Rover performed by NASA/JPL's Laboratory for Reliable Software [21]. The missing coverage was mostly defensive, redundant code, including fail-safe termination after (presumably impossible without hardware failure) assertion failures.

7.2 Improving Bitcoin Core Fuzzing

The lack of a significant coverage gap, and the existence of a huge mutant-detection gap, for transaction verification, between the fuzzing and the functional tests suggests the most promising method for improving the Bitcoin Core fuzzing: manual, expert developer effort to improve the oracles used by existing fuzz targets, or efforts to craft custom, more restricted, fuzz targets with stronger oracles when this is not feasible.

Essentially, the lack of a huge coverage gap suggests that failure to explore the input space is not the biggest problem for the Bitcoin Core fuzzing; there's significant room for improvement, especially with some targets, but overall the coverage, especially for critical code, is good. What the mutation testing reveals is that many fuzz executions that would be rejected by the standards of the functional tests do not induce a crash.

Building fuzz harnesses with complex correctness checks is hard, of course; the functional tests know exactly what inputs are being provided to APIs, and can check for expected behavior. Trying to inject this kind of check into fuzz harnesses ranges from non-trivial to effectively impossible (for some properties). When applicable, more generic, "mathematical" constraints such as are used in property-based testing [9] can help, but these are often hard to apply at the end-to-end level of a fuzz harness, without turning the fuzz harness into a spaghetti code mess of checks for various qualities of the inputs. In the worst case, these end up simply reflecting developer/tester notions that already found their way into the code itself. Differential testing [29] against other implementations might also help here, but is probably best done at a higher end-to-end level than inside these specific fuzz targets (this is already done on some cryptographic elements of the code).

There is no easy solution to this problem, but the most promising route is probably to focus on adding *invariants and assertions* to the non-test code itself; these checks can be executed by both functional and fuzz tests, and avoid the problem of duplicating analysis of inputs. At present, the Bitcoin Core code has about 1,800 `assert` statements, scattered among 180KLOC of C and C++. The resulting ratio of about one assertion per 100 lines of code is not terrible, but is at the lower limit of what many consider to be an acceptable assertion ratio for critical code. Given that Bitcoin Core defines at least 4,000 functions, the code obviously doesn't meet the NASA/JPL proposal of having an average of two assertions per function [25]. There are only five assert statements in the `src/consensus` directory, which has about 500 lines of code and defines more than 10 functions, suggesting that the assertion ratio is low even for critical code.

One conclusion of the 80 hour effort therefore, is that the most effective way to improve fuzzing at present might be not to focus on covering the code and state space, but to focus on increasing the oracle power of *all* Bitcoin Core testing. Arguably, the greatest weakness of traditional code coverage is that it focuses too much attention on the "input side" of testing and too little on the oracle side [3], which is easier for even dedicated testing efforts to neglect in the pursuit of covering every branch and path.

Acknowledgements: The authors would like to thank the Chaincode team, particularly Adam Jonas and Evan Baer, and especially the "fuzzing gurus" for Bitcoin Core, `MarcoFalke` and `practicalswift` on GitHub.

REFERENCES

[1] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. 2016. Can Testedness Be Effectively Measured?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 547–558. https://doi.org/10.1145/2950290.2950324

[2] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. 2017. Applying Mutation Analysis on Kernel Test Suites: an Experience Report. In *International Workshop on Mutation Analysis*. 110–115.

[3] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[4] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 713–724. https://doi.org/10.1145/3368089.3409729

[5] Timothy Budd, Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1979. *Mutation analysis*. Yale University, Department of Computer Science.

[6] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/2491956.2462173

[7] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *USENIX Security Symposium*. 1967–1983.

[8] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*. 736–747.

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*. 268–279.

[10] Thomas G Dietterich et al. 2002. Ensemble learning. *The handbook of brain theory and neural networks* 2 (2002), 110–125.

[11] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830–840. https://doi.org/10.1145/3338906.3338945

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[13] Teofilo F. Gonzalez. 1985. Clustering to Minimize the Maximum Intercluster Distance. *Theoretical Computer Science* 38 (1985), 293–306.

[14] Peter Goodman and Alex Groce. 2018. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*.

[15] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 72–82. https://doi.org/10.1145/2568225.2568278

[16] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. 2018. How verified (or tested) is my code? falsification-driven verification and testing. *Automated Software Engineering Journal* 25, 4 (2018), 917–960.

[17] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and Its Discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward2014)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/2661136.2661157

[18] Alex Groce and Martin Erwig. 2012. Finding Common Ground: Choose, Assert, and Assume. In *International Workshop on Dynamic Analysis*. 12–17.

[19] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. 2020. What are the Actual Flaws in Important Smart Contracts (and How Can We Find Them)?. In *International Conference on Financial Cryptography and Data Security*. 634–653.

[20] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-expression-based Tool for Multi-language Mutant Generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/3183440.3183485

[21] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing as a Prelude to Formal Verification. In *29th International Conference on Software Engineering (ICSE'07)*. 621–631. https://doi.org/10.1109/ICSE.2007.68

[22] Alex Groce, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Yang Chen, and John Regehr. 2013. Help, help, I'm being suppressed! The significance of suppressors in software testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 390–399.

[23] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *International Symposium on Software Testing and Analysis*. 78–88.

[24] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. 2020. Using Relative Lines of Code to Guide Automated Test Generation for Python. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 28 (Sept. 2020), 38 pages. https://doi.org/10.1145/3408896

[25] Gerard J Holzmann. 2006. The power of 10: Rules for developing safety-critical code. *Computer* 39, 6 (2006), 95–99.

[26] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices* 49, 6 (2014), 216–226.

[28] Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[29] William McKeeman. 1998. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation* 10(1) (1998), 100–107.

[30] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

[31] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[32] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *ACM SIGMOD*.