2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 978-1-6654-9590-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-SEIP55303.2022.9794086

# 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. This paper describes an effort to investigate and enhance the effectiveness of the Bitcoin Core 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. This paper summarizes the outcomes of a two-week focused effort. While the effort found no smoking guns indicating major test/fuzz weaknesses, it produced a large number of additional fuzz corpus entries, increased the set of fuzzers used for Bitcoin Core, and ran mutation analysis of Bitcoin Core fuzz targets, with a comparison to Bitcoin functional tests and other cryptocurrencies' tests. Our conclusion is that for high quality fuzzing efforts, improvements to the *oracle* may be the best way to get 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. 2022. Looking for Lacunae in Bitcoin Core's Fuzzing Efforts. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3510457.3513072

## 1 INTRODUCTION

Bitcoin [8] is the most popular cryptocurrency, and, while volatile, has a market cap consistently over half a trillion dollars since January of 2021. Bitcoin Core (https://github.com/Bitcoin/Bitcoin) is by far the most popular implementation, and serves as a reference for all other implementations of Bitcoin. To a significant degree, the code of Bitcoin Core *is* Bitcoin. Because of its fame and the high value of Bitcoins, Bitcoin is a high-value target for hackers. 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.

Chaincode Labs (https://chaincode.com/) is a private R&D center that exists solely to support and develop Bitcoin. In March of 2021 the head of special projects at Chaincode contacted the first author to discuss determining a strategy to improve the fuzzing of Bitcoin Core. It seemed that the fuzzing was "stuck": neither code coverage nor found bugs were increasing with additional fuzzing. After some discussion, an 80 hour effort was determined as a reasonable scope for an external, research-oriented, look at the fuzzing effort.

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... [and] the number of new bugs found by the fuzzer drops off, eventually approaching zero." At first a particular fuzzer applied to a system will tend to continuously 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 complex probability distribution. Some bugs lie in the high-probability portion of this space, and other bugs lie in very low probability zero probability parts. Escaping saturation may require a variety of approaches.

## 2 RESULTS

One thing that quickly emerged from discussions before the primary 80 hour effort began was the limited extent of the fuzzer runs being performed. 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. For complex targets such as transaction verification and end-to-end message processing, 100,000 iterations was highly insufficient. The first suggestion for escaping saturation, therefore was very simple: run the fuzzer longer! The Chaincode tried increasing their configuration to 5 million iterations, multiplying the number of

*ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513072

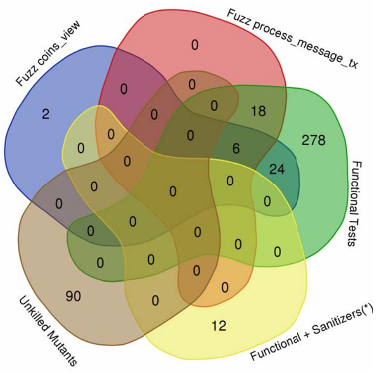

**Figure 1: Mutation kills for `tx_verify.cpp`**

executions by a factor of 50. Based on initial success with a few targets, this was done for all targets, and became the new default. By May 20th, Bitcoin Core was also in OSS-Fuzz: 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.

Additonal efforts to improve the fuzzing focused on adding support for the Eclipser fuzzer [2], and attempting to use swarm testing [6] to produce more unusual message sequence interactions. Using Eclipser produced a large number of additional corpus seeds for OSS-Fuzz (over 2,000 inputs, the third largest contribution to the set to date), while swarm turned out to be ineffective, due to extensive manual cross-seeding providing similar benefits. Neither approach, however, turned up any new bugs and improvements were in some sense marginal (new paths or data values only).

## 3 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 exposed a serious problem. 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 [9], however, subsumes code coverage and adds extremely valuable information on *oracle power* in addition to mere coverage [4]. 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 [1, 3].

We used the Universal Mutator (https://github.com/agroce/universalmutator) [5] to mutate transaction verification code in the `tx_verify.cpp` file; this is clearly extremely critical functionality. 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. The Universal Mutator generated 430 mutants. The `process_message_tx` fuzz 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). Fuzzing could detect just under 12% of mutants. Fuzzing adds only two unique mutant kills beyond those produced by functional testing, which has a much higher score. This raises the question: why fuzz? 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.

To put Bitcoin Core in context, we performed mutation analysis of transaction-verification-related code for other cryptocurrencies, and Bitcoin ranked high: 2nd out of 6 projects. Bitcoin Core also had the highest File and Project coverage of any project.

Our conclusion, based on the negligible gap between code coverage for fuzzing and functional tests, and the huge difference in mutation scores, and the lack of new bugs found even when we ran novel fuzzers, is that the best way for Bitcoin to gain fuzzing power might be to improve the oracle power in fuzzing by adding more invariants and sanity checks. 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 [7]. 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 very critical code.

**Full Report:** The full report on this effort is available at https://agroce.github.io/bitcoin_report.pdf. **Acknowledgements:** A portion of this work was supported by the National Science Foundation under CCF-2129446; the authors would also like to thank Chaincode Labs and the Bitcoin Core team.

## REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and Its Discontents. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. 255–268.

[5] 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

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

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

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

[9] 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.