Looking for Lacunae in Bitcoin Core's Fuzzing Efforts

Alex Groce
Northern Arizona 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 OSSFuzz triage, increased the set of documented fuzzers used in Bitcoin Core testing, and ran the first mutation analysis of Bitcoin Core code, revealing opportunities for further improvement. We contrast the Bitcoin Core transaction verification testing with similar tests for the most popular Ethereum and dogecoin implementations. This paper provides an overview of the challenges involved in improving testing infrastructure, processess, and documentation for a highly visible open source target system, from both the state-of-the-art research perspective and the practical engineering perspective.

CCS CONCEPTS

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

KEYWORDS

fuzzing, saturation, test diversity, mutation analysis

ACM Reference Format:
Alex Groce. 2021. Looking for Lacunae in Bitcoin Core's Fuzzing Efforts. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '21), October 20–22, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3486607.3486772

1 INTRODUCTION

Bitcoin [20] 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 degreee, 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.

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 fuzzer 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-probabiliity portion of this space, and other bugs like in very low probability (or in some cases, 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" [2].

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 We Hold the World Ransom For... ONE MILLION FUZZER ITERATIONS

One problem for inexperienced fuzzer users is that it is not always clear just how long a fuzzer run is needed. Anyone used to popular random testing tools, such as QuickCheck [6], may expect a "reasonable" budget to be on the order of hundreds of tests or at most a few minutes [17]. The fuzz testing research community has settled, to some extent, on 24 hours as a basis for evaluation of fuzzers [18], 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*.

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 as inadequate a Dr. Evil's famous ransom demand for "one million dollars" in the first Austen Powers movie. Neither the Chaincode team nor Dr. Evil was being unreasonable; they were both simply unaware of the "inflation" of needed resources required for serious fuzzing or blackmail of the world.

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 increasinig 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 faces saturation on a target codebase is to bring in fuzzer B. 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 bettter.

3.1 Fuzzing with Eclipser

Eclipser [5] is a fuzzer that combines afl-like[1] coverage-driven mutation-based fuzzing with a scalable form of grey-box concolic testing. AFL++ [8] 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.

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, 11]. 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.

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

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 [4] 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 [7], 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 [10] front-end[2].

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 verson not intended to be rolled into the main codebase 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.

4 SIDE ISSUES: TRIAGE, SPACE USAGE, AND FRUSTRATING BUGS

5 TRYING SWARM FUZZING

Swarm testing [16] is a method for improving test generation that relies on identifying *features* [15] 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 disable 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 [19] and is a core element of the testing for FoundationDB [22].

Some Bitcoin Core fuzz harnesses resemble API-sequence generation. In particular, the `process_messages` produces method sequences of a fixed length, where each message is of a particular type (there are a modest number of message types, again similar to an API-call "fuzz surface"). Swarm testing is an obvious candidate approach for enhancing this important fuzz harness.

DeepState [10] provides strong support for swarm fuzzing. However, as with ensemble fuzzing, while there are similarities to the Bitcoin Core infrastucture for fuzz targets and the DeepState API, re-writing the fuzz targets to use DeepState seemed like 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 bitstreams to find commands is hard, and writing a dynamic parser to use in-filght 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. 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 fixing the type string. The ability to run `process_message` with a single message type and the frequent introduction of resulting data into `process_messages` 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 han unclear how many fuzz frameworks are as

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

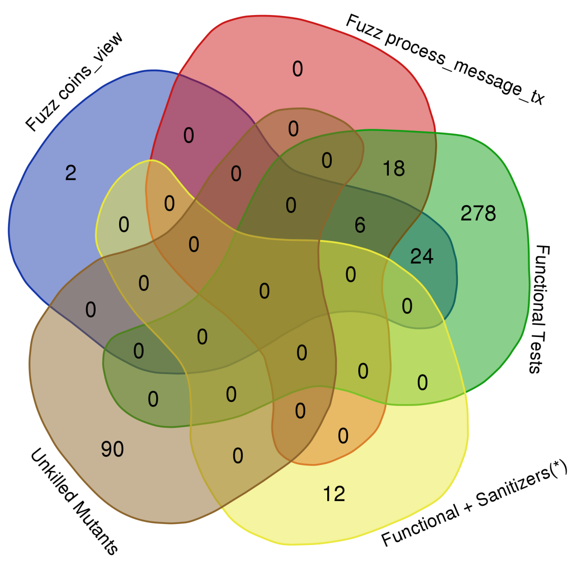

Figure 1: Mutation Kills for `tx_verify.cpp`

sophisticated as Bitcoin Core's, making this possible; in many cases simply implementing swarm directly would likely be easier. It does show that some advanced fuzzing strategies can be anticipated by particularly savvy and capable fuzz effort engineers, willing to directly use raw fuzzing data, and write tools to support that kind of low-level hand-tuning of fuzz corpuses.

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 exposed a serious problem. Adding more fuzzers would be *good*, but waas not obviously *essential*. 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, so little value would be added by inspecting mere coverage. Mutation testing/analysis [21], however, subsumes code coverage and adds extremely valuable information on *oracle power* in addition to mere coverage [12]. This is perhaps especially valuable in fuzzing, where "you only see crashes" is a persistent concern.

We used the universal mutator (https://github.com/agroce/universalmutator) [14] to mutate the Bitcoin Core `tx_verify.cpp` code. Transaction verification is a well-covered (https://marcofalke.github.io/btc_cov/fuzz.coverage/src/consensus/index.html) file, and obviously an extremely critical functionality for the blockchain. Figure 1 shows an overview of the mutation analysis of the file.

The universal mutator generated 430 compiling mutants of the file. Two fuzz targets performed interesting testing of the `tx_verify.cpp` code: `process_message_tx` and `coins_view`. The code was also tested by `process_message` and `process_messages` but the relevant corpus entries were duplicated in `process_message_tx` (we verified these provided no additional mutant kills). The fuzzing applied was limited to five minutes of libFuzzer exploration based no the full (and large) 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, not quite 12% of all the generated mutants. This is not 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 [13]. 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 (which is very expensive — we *only* ran it for mutants surviving all other tests, as indicated by the zero overlap and the asterisk in Figure 1) catches an additional 12 mutants. Only 90 of the 430 compiling mutants survive all tests, for an overall mutation score of 79.07%. On the other hand, fuzzing only adds two mutant kills to the functional testing.

Moreover, manual inspection of the 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 mutants, prioritized by an FPF ranking [3, 9], 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 the surviving 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.

6.1 Comparison with Other Cryptocurrency Codebases

We also use mutation to (very loosely) compare Bitcoin Core's testing with that for other popular cryptocurrencies. Again, we targeted transaction validation logic, for the same reasons as given above. To select our corpus, we examined the top 10 cryptocurrencies by market cap. We eliminate stable cryptocurrencies, such as USD coin, because they do not have transaction validation logic analagous to bitcoin. For each project, we also run coverage collection tools in order to compare code coverage. If a project does not build or have coverage, we also exclude it from our comparison.

Project	File Path	Mutation Score	Coverage
bitcoin	src/consensus/tx_verify.cpp	78.6%	98.7%
go-ethereum	signer/fourbyte/validation.go	49.5%	60.0%
dogecoin	src/bitcoin-tx.cpp	58.7%	
avalanchego	vms/platformvm/add_subnet_validator_tx.go	57.3%	81.0%
stellar	src/historywork/VerifyTxResultsWork.cpp	85.1%	

Table 1: Comparison of Coverage and Mutation Score Across Projects
rvt asks: for the reader, what does mutation score represent? (feel free to explain inside this caption)
rvt says: probably we only need the filename(s) and not the path for presentation–will save space and this table could fit into one col. small potatoes right now though.
rvt asks: Coverage here represents coverage of the file–I think it would be good to provide the average coverage per project too, to give an overall sense for the project. We should caveat in the discussion, what this number means, i.e., it is representative only of the subset of things we did to try determine coverage, and not necessarily an accurate signal of testing quality, because we might miss, e.g., integration test coverage or I don't know what. So this is more to satisfy a "here's the data if you want to form a picture in your head of testing as per what we observed"

In the end, the following projects failed to compile: Binance Coin, Polkadot, Chainlink and the following projects did not have readily accessable coverage reports: Stellar.

In deciding what files to mutate, we select the core transaction validation logic, by filtering for files with tx, transaction, verify, and validate. Following this, we manually inspected each matching file and select ones that directly relate to transaction validation. The results for these experiments are displayed in Table 1.

7 HOW GOOD IS THE BITCOIN CORE TESTING AND FUZZING?

8 CONCLUSIONS

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] 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

[3] 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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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

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

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

[11] 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

[12] 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

[13] 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.

[14] 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

[15] 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.

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

[17] 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

[18] 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

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

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

[21] 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.

[22] 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*.