Evaluating Fuzzing Changes for Long-Running Fuzzing Campaigns

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Best practices for evaluating new fuzzing algorithms have largely been established in a research-centric context: fuzzers are compared by running them repeatedly over a set of benchmarks, and comparing code coverage (and, to some extent, bug detection) from an intially empty or minimal corpus. While reasonable for evaluating fuzzing research, such methods are not useful in the context of a long-running fuzzing campaign. E.g., a major open source project using Google's OSS-Fuzz may have an estabilshed corpus of tens or hundreds of thousands of inputs; a method that performs well on an empty corpus may be incapable of improving on the results of a vast number of CPU hours spent fuzzing. Evaluating using the established corpuses, however, is likely to show that all proposed methods fail to discover new coverage elements in a reasonable amount of evaluation time. We propose a practical method, based on binary-level mutation, that addresses these problems. In part our approach relies on a novel distinction between *oracle* improvements and *fuzzer efficiency* improvements. The approach is primarily evaluated by real-world use on the extensive fuzzing infrastructure for the bitcoin core implementation, and by smaller-scale experiments over a set of additional large, heavily-fuzzed, open source systems.

1 Introduction

Software, of course, changes over time; this means that even an ostensibly bug-free program, if important enough to matter, must be supported by a regime of tests that determine if future changes induce new bugs. For critically important software facing not only "natural" disasters of unexpected input sequences, but human adversaries aiming to undermine the system, the level of testing required to protect software is extremely high indeed. *Fuzzing* [12], the subjection of a program to randomly generated (and, usually, intelligently adapted based on code coverage and other signals) inputs, is one of the most important tools for assuring that such programs are secure and reliable. To this end, the most critical software systems are often subjected to computationally vigorous long-running *fuzzing campaigns*. E.g., Google's oss-fuzz service [1] provides continuous fuzzing of over a thousand critical open-source software systems, and has detected more than 10,000 vulnerabilities and 30,000 bugs, to date.

Fuzzing efforts, too, change over time. When new functionality is introduced to a software system, new tests must often be added to the code that fuzzes a system, especially when fuzzing is implemented as a series of independent fuzzers targeting particular functionalities, rather than a single monolithic subjection of a whole program to fuzz inputs. Fuzzing engineers also introduce changes to how fuzzing is performed, reconfiguring the generation and adaptation of inputs to reflect improvements to the art and science of fuzzing, and to avoid *saturation*. Saturation [8, 11] is an effect where, even if in theory a particular fuzzing approach is highly effective, it over time discovers all interesting program inputs for which *this particular approach* is highly effective, and can be improved upon even by less (theoretically) effective approaches that have an easier time reaching new behaviors. Furthermore, fuzzing engineers often try to improve the power of a fuzzing effort by adding new checks for correctness to a fuzzing effort, adding to the set of behaviors that are deemed to have exposed a bug or vulnerability in the system.

Like software development, fuzzing development is subject to human error; an intended improvement to a fuzzing configuration can actually degrade the ability to detect bugs. Furthermore, fuzzing is subject to surprises that are not rooted in human error: a change that should, so far as human understanding is concerned, be an improvement may, for reasons beyond human understanding (given the complexity of fuzzing and our current understanding of software systems) turn out to be harmful instead of beneficial. Thus, like developers of production code, developers of fuzzing code and configurations need a way to determine if their efforts are beneficial or harmful to the task at hand. Fuzzing engineers need ways to evaluate changes in fuzzing effectiveness, just as software developers need ways to check for bugs in the software they are developing.

How do fuzzing engineering go about this problem? The evaluation of fuzzers is a topic of substantial study in the literature, ranging from summations of widely agreed-upon conventions [9] to highly speculative approches [6]. In general, these evaluations are all based on the idea of comparing fuzzers by computing statistics over evaluation measures collected for multiple runs of each fuzzer over a set of benchmarks. Evaluation measures are typically limited to code coverage and counts of distinct bugs detected (the latter often somewhat approximate [4]). These evaluations almost always "start from scratch" and determine, essentially, how well fuzzers can fuzz a previously un-fuzzed program in a period of 24-48 hours.

This approach may be useful for evaluating the promise of novel fuzzing algorithms in academic research papers, but it has significant limitations in evaluating proposed changes to long-running fuzzing campaigns:

1. First, and most importantly, starting from scratch (an empty corpus of previously generated interesting inputs) is inappropriate. Engineers do not want to know how well fuzzing would go if they threw away the results of many thousands of hours of compute time spent exploring a program. Starting from scratch, for example, completely eliminates the effects of saturation. Appropriate evaluations of real-world long-running fuzzing campaigns must generally start from the existing corpus of discovered inputs.

2. Given this fundamental difference, basing evaluations on incremental code coverage is almost guaranteed to be impossible, if the evaluation time is limited to standard 24 hour runs, or even to 48 hour runs. In fact, for stable programs that have been long-fuzzed in a setting such as OSS-Fuzz, reaching new coverage may require weeks of fuzzing, or coverage may essentially be total, especially for critical parts of the program (e.g., the Bitcoin Core fuzzing reached essentially total coverage of transaction validation years ago [7]).

3. Similarly, expecting new methods (or old ones) to find bugs in programs that have been fuzzed for many thousands of hours in a 24 or 48 hour time period is highly unrealistic. In practice, both coverage and bug counts, incremental from a corpus of inputs collected over what may be years of fuzzing, will be almost always be *zero* for all methods.

Given these problems with the standard evaluation methods, how can engineers trying to improve a long-running fuzzing effort evaluate their proposed changes? Fundamentally, the problem is that there just are not enough coverage targets and real bugs to form a basis for decision-making.

1.1 Evaulation of Fuzzer Changes via Program Mutants

Mutation testing is, essentially, the evaluation of testing efforts by the injection of "fake bugs." While introduced in the late 1970s [10,10,13], it has recently been aggressively adopted by major software developers, including Google [16], Meta [3], and Amazon [15], as the technology for mutation testing has improved and sufficient computing power to make it practical has become available. Mutation testing relies on mutation generation, the production of small random changes in a program; such changes are expected to usually introduce bugs in the program (given the original code was correct). Mutation generation is available for essentially all even moderately-widely-used programming languages [5].

Mutation testing provides a way to get around the problem that there are too few coverage targets and too few real, undetected, bugs in software to enable effective evaluation of changes to real-world fuzzing campaigns. Mutations provides a source of large numbers of program behaviors that are of interest (in that the represent *potential* bugs in software). While most program mutants are likely to be of little interest (easily detected by existing tests/fuzzing corpus), the large number of mutants for even a small part of a program (on average, perhaps as many as 2 mutants per LOC) means that many mutants that represent hard-to-detect bugs are likely to exist for any program of substantial size.

In order to effectively evaluate changes to long-running fuzzing campaigns, we make a core distinction between two kinds of changes. The first kind, *oracle changes*, can be evaluated using what are essentially off-the-shelf mutation testing techniques, properly configured. The second type of changes, *fuzzing strategy changes*, however, require a set of complex changes to standard mutation testing practice in order to evaluate changes in a time-effective manner.

1.2 Evaluating Oracle Changes

Oracle [2, 17] changes are those modifications to a fuzz target or fuzz harness whose purpose is to *increase the set of executions that are deemed test failures*; in fuzzing, this basically means changes that *increase the number of crashing inputs.* Oracle changes range from adding a single, highly specific, assertion inside SUT code, to extensive rewriting of a fuzz harness, potentially changing the set of inputs that it is capable of generating and adding an expensive and complex check for correct execution after the change. An example of the latter is transforming a normal fuzz harness into a *differential* [14] harness, where inputs are applied to both the SUT and a reference implementation, and their behavior is compared (with certain differences considered as test failures).

At first glance, it may seem that no evaluation is needed for oracle changes: increasing the set of failing runs is always good, unless the change introduces a large number of false

positives. If a change introduces few false positives, and many true positives, it is good, and no special method is needed to observe that a change has produced so many false positives it is making the fuzzing campaign less useful. The problem is that an oracle change may have costs other than false positives. In particular, some oracle changes greatly reduce fuzzing throughput. Differential oracles, while very powerful, often at least double the time to run a particular input, and may be more costly than that, if the reference implementation is slow or the comparison of behaviors is complex. If an oracle change detects few bugs, and greatly reduces the ability of the fuzzer to explore behaviors, it may be harmful, or at least best limited to occasional runs over a corpus produced by more efficient methods.

2 Background and Terminology

3 Evaluation for Long-Running Campaigns

4 Experimental Evaluation

In order to evaluate our approach, we studied its performance on past changes made to open source projects using Google's OSS-Fuzz infrastructure. In particular, we selected projects:

- With a long history of OSS-Fuzz usage, so we could ensure we were applying our technique to the target class of problem: changes to fuzzers with a long history and mature/potentially saturated corpus.
- Such that we could extract past corpus data from revision control or other sources, and apply evaluations of changes to the appropriate state of the campaign.

Projects with a long history of OSS-Fuzz usage are essentially guaranteed to be critical software with large real-world impact and it is generally the case that their fuzzing efforts are well-supported (hence early adoption of OSS-Fuzz). The one exception to the rule of preferring projects with longer OSS-Fuzz history was that we also chose some changes from the Bitcoin Core implementation, with the guidance of Bitcoin Core fuzzing engineers. This allowed us to include changes specifically identified as of interest to fuzzing engineers on a highly critical project who were considering adopting our technique.

For each project and each change (see Table **??**) we applied multiple evaluation methods. Each evaluation was applied on a basis of 10 runs of 48 hours for each run. We evaluated every change by the following baseline methods:

- Statement coverage
- Branch coverage
- Crash count

We did not attempt to disambiguate bugs beyond the standard crash bucketing measures of fuzzers. Additionally, depending on whether we classified a change as an oracle change or a fuzzer efficiency change, we applied the respective relevant technique ("pure" mutation testing and testing for re-discovery of observable mutants) proposed in this paper. We selected changes with a bias towards fuzzer efficiency changes, as the proposed method is more complex and non-standard, and in our opinion thus requires more experimental evidence of its effectiveness. The discussion of results below further explains how which statistics of the 10 runs for each evaluation we used to examine the sensitivity of methods to changes in fuzzer performance.

4.1 Research Questions

- RQ1: Are our methods consistently more capable of detecting fuzzer changes than the baseline methods?
- RQ2: How often do our methods fail to detect that a change has an impact?
- RQ3: In cases where fuzzer changes were rolled-back in a project (indicating either a negative impact on fuzzing or at least a lack of positive impact worth preserving) were our methods able to predict this outcome?

5 Conclusions and Future Work

References

[1] oss-fuzz. https://github.com/google/oss-fuzz.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[3] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - A study at facebook. In *International Conference on Software Engineering: Software Engineering in Practice*, pages 268–277. IEEE, 2021.

[4] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[5] Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. Syntax is all you need: A universal-language approach to mutant generation. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[6] Miroslav Gavrilov, Kyle Dewey, Alex Groce, Davina Zamanzadeh, and Ben Hardekopf. A practical, principled measure of fuzzer appeal: A preliminary study. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 510–517, 2020.

[7] Alex Groce, Kush Jain, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. Looking for lacunae in bitcoin core's fuzzing efforts. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 185–186, 2022.

[8] Alex Groce and John Regehr. The saturation effect in fuzzing. https://blog.regehr.org/archives/1796.

[9] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. ACM.

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

[11] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. Reachable coverage: Estimating saturation in fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 371–383, 2023.

[12] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[13] Aditya P Mathur. *Foundations of Software Testing*. Addison-Wesley, 2012.

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

[15] Giorgio Natili. Mutation testing at scale. https://slides.com/giorgionatili/mutation-testing-at-scale.

[16] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 47–53, 2018.

[17] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 391–400, New York, NY, USA, 2011. ACM.