On the Influence of Compiler Optimizations on Fuzzing: Proposal

ANDRIN BERTSCHI, Automated Software Testing, ETH Zurich, Switzerland
MATTHEW WEINGARTEN, Automated Software Testing, ETH Zurich, Switzerland

1 INTRODUCTION

Fuzzing has become an integral part to detect security bugs in software. Fuzzing is an automated software testing technique where a fuzzer generates plenty of test data for a target program with the aim to trigger a software bug [Liang et al. 2018]. Practitioners typically fuzz binaries compiled with default compiler optimizations such as -O2, or -O3[1]. Binaries compiled from the the same source code may vastly differ in control flow. We suspect differences in control flow graphs of binaries may have an impact on the performance of fuzzing. We aim to analyze the impact of different compiler optimizations on fuzzing and provide further insight.

Specifically, the questions we set out to answer are as follows:

- Do optimization flags have an impact on fuzzer performance, specifically edge coverage and bug finding rate?
- If so, how disjoint are the produced test cases in regards to edge coverage?
- How much performance gain can we get from combining multiple fuzzing runs on different output binaries?
- Can we estimate the difference in fuzzing performance based on binary code differences?
- If we can achieve a performance increase, can we find either a minimal set of optimization flags or binary generation frameworks to maximizes coverage? Or can we find a set of flags that produce a single binary that maximize performance?

2 RELATED WORK

Controlled Compilation has been shown to improve fuzzing performance [Simon and Verma 2020]. This study argues that binaries optimized for runtime performance are poor for fuzzing performance. The reason is, coverage-guided grey-box fuzzers, such as AFL, rely on code coverage metrics to guide input generation, giving insight into which parts of the program have been reached. Optimizations that *reduce* branches give a fuzzer less feedback to work with. More specifically, optimizations that reduce the complexity of the control flow graph can harm fuzzing performance. These results should be kept in mind when choosing which optimization passes we want to analyze. We wish to further explore these concepts.

Furthermore, InstruGuard shows that different optimization flags can affect the fuzzing performance, although not directly caused by binary differences [Liu et al. 2021]. They show that optimization flags can break the instrumentation, for example, missed instrumentation locations, and they implement a tool to detect and repair such instrumentation errors with binary rewriting. The results from this paper are important to keep in mind to answer our research questions, not to draw the wrong conclusion about the root cause of fuzzing performance (whether that is instrumentation bugs or actual underlying binary difference). It may be worth running InstruGuard on our generated binaries to rule out changes in performance caused by instrumentation errors.

[1] cf. Optimization Options in gcc https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options

Authors' addresses: Andrin Bertschi, andrin.bertschi@inf.ethz.ch, Automated Software Testing, ETH Zurich, Switzerland; Matthew Weingarten, wmatt@student.ethz.ch, Automated Software Testing, ETH Zurich, Switzerland.

3 APPROACH

For this work, we use AFL++ [Fioraldi et al. 2020], the Clang and LLVM compiler and its optimization flags, and BinTuner [Ren et al. 2021] as a binary code generator and analysis tool. Furthermore, we will sample three test cases from fuzzing bench suites: Fuzzbench [Fuzzbench 2020], Magma [Hazimeh et al. 2020], and Unifuzz [Li et al. 2021].

For every benchmark/optimization flag pair, we have the following pipeline:

- Do multiple fuzzing runs with fixed time on each pair and generate a test set.
- Using the test set of a benchmark/flag pair we determine edge coverage and bugs found. Performance metrics can only be evaluated and compared to a particular binary. This suggests we compute and compare metrics for every benchmark/flag pair test set on every benchmark/flag binary.
- We quantify binary difference with a binary code analyzer (i.e BinTuner).

We expect the aforementioned pipeline to give us enough data to answer most questions in our problem statement. Based on our analysis we will further try to expand on current AFL++ performance, i.e find a minimal set of optimization flags and/or use BinTuner to further increase fuzzing performance.

4 WORK SCHEDULE

By progress report (due 29.04):

- Finish script for pipeline
- Run all experiments for basic flags: -O0, -O1, -O2, -O3
- Have a great understanding of BinTuner (or other binary code analysis tools). Formalize how we quantify binary difference and how we aim to generate code with a high binary difference.
- Decide which single optimization flags are interesting to experiment with (for example loop unrolling).
- Decide whether we should use other performance metrics apart from edge coverage (i.e bugs found with Adresssanitizer or Valgrind)

By final report (due 6.6):

- Expand our analysis of experiment results. Visualize it well with plots. Formalize any noticed patterns.
- Run all experiments on interesting single optimization flags.
- Present answer to all questions asked in our problem statement.

REFERENCES

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. *AFL++: Combining Incremental Steps of Fuzzing Research.* USENIX Association, USA.

Fuzzbench. 2020. Fuzzbench report. https://www.fuzzbench.com/reports/sample/index.html

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. https://doi.org/10.1145/3428334

Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *Proceedings of the 30th USENIX Security Symposium.*

Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476

Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. 2021. InstruGuard: Find and Fix Instrumentation Errors for Coverage-based Greybox Fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 568–580. https://doi.org/10.1109/ASE51524.2021.9678671

Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study. *CoRR* abs/2103.12357 (2021). arXiv:2103.12357 https://arxiv.org/abs/2103.12357

Laurent Simon and Akash Verma. 2020. Improving Fuzzing through Controlled Compilation. In *2020 IEEE European Symposium on Security and Privacy (EuroS P).* 34–52. https://doi.org/10.1109/EuroSP48549.2020.00011