On the Influence of Compiler Optimizations on Fuzzing: Status report

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

1 CURRENT STATE OF THE PROJECT

Currently, we have built an experiment pipeline from compiling benchmarks, fuzzing binaries, to generating data and plots. We ran eight different benchmarks with compiled with Clang flags *-O0, -O1, -O2, -O3*. We use source-code based edge coverage as performance metric. For each benchmark/flag pair we have run 3 trials for 4 hours, as less fuzzing time led to high variance in fuzzing performance.

Fuzzing Pipeline. We use a fork of AFL++ as our fuzzer with the clang/LLVM backend. We modified AFL++ to accept additional compiler flags for the instrumentation. The compiler flags can be set as an environment variable as illustrated below.

```
AST_CC_ARGS="-O0 -loops" \
AFL_DONT_OPTIMIZE=1 afl-cc <binary>
```

We try to follow a realistic AFL++ configuration and use a dictionary, *persistent mode*, *Fast LLVM-based instrumentation (afl-clang-fast)*, and *CmpLog* to overcome checksums and magic bytes. These configurations were used in [Fuzzbench 2020] as a baseline for an AFL++ based fuzzer.

We use the binary analysis tool Angr [Shoshitaishvili et al. 2016] to analyze the instrumented binaries produced by different flags (e.g. to compare the number of edges of the binaries produced by different flags). Furthermore, we forked Fuzzbench introduced by [Fuzzbench 2020] and extended it with additional fuzzers to suit our needs. This way we can use the Fuzzbench as a fuzzing-as-a-service model. Critically, Fuzzbench already provides plumbing code to clone, build and instrument well-fuzzed benchmarks, with default input seeds. It is, however, optimized for the cloud[1] Additionally, Fuzzbench employs the *LLVM Source-based Code Coverage*[LLVM 2022] tool to evaluate fuzzing performance, which means we can obtain edge coverage based on the original source code of a benchmark and not its binary representation. Otherwise, when comparing edge coverage of different optimization flags, it becomes unclear which binary to evaluate the fuzzer's performance on and may skew results. We extend Fuzzbench's visualization code to create plots which aid in our decision making. While this does not cover everything we wish to evaluate, it allows us to data-driven decisions without extensive overhead, including statistical significance tests.

In a nutshell, we now can now easily queue experiments to run over night, aggregate data and compare performance of different compiler optimization flags. We have the capability to perform binary analysis with CFG reconstruction for more complex analysis.

[1]For each fuzzing execution, a docker image is built which causes overhead when run on a local machine.

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

2 MEASUREMENT RESULTS AND DATA

The set of benchmarks we have run so far are: *bloaty fuzz target*, *freetype*, *lcms*, *libpng*, *libxml*, *openthread*, *sqlite3*, *vorbis* and *woff2*. Our preliminary results suggest that there is *no* statistical difference between the four -O levels. This is further identified in Figure 1. Figure 2 depicts the absolute edge coverage between the different optimization flags across all benchmarks. We see only small (insignificant) differences in edge coverage across the different flags. The hypothesis that the performance of a single fuzzer is better is almost always rejected. Figure 4 and Figure 3 display a violin plot and a Mann-Whitney test, respectively. While we only list a few benchmarks here, they are representative for every benchmark of our current preliminary results. Refer to Section 4 for more sample of plots we plan use in the final report.

3 HOW TO PROCEED

Recall the research questions from our original proposal:

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

To answer (1) and (4) we must dive deeper: the -O level flags currently do not show to have a big difference on fuzzing performance. However, this reinforces observations made in [Simon and Verma 2020]. We hypothesize there that is a delicate trade between runtime performance (faster fuzzing allows for more runs), and code complexity. Some optimizations to decrease runtime may harm the coverage-based feedback system of AFL++. We will further examine this using data gathered in the form of Table 1 and analyze single flags in isolation. For example *–simplify-cfg* might be a good flag to turn off, as it may cause the fuzzer to get less feedback from edges covered. With our fuzzing-pipeline we are not able to iteratively gather new experiment data.

Questions (2) and (3) go hand in hand. We argue the more disjoint regions two flags cover, the higher chance combining two flags could outperform running with a single flag. It is possible that a specific region is hard to reach with certain flags, and running with two different binaries may alleviate this issue. We propose the following experiment to test this theory: Run a fuzzing-binary with flags *f1* for half the benchmark time, take its corpus and feed it into the second fuzzing-binary with flags *f2*. We choose f1 and f2 based on results from data gathered in the form of table Table 2, such that f1 and f2 have the highest difference in unique regions covered across all benchmarks.

Answering (5) has currently the lowest priority: We quantify binary difference with a binary code analyzer (i.e BinTuner [Ren et al. 2021]) and try to obtain a set of compiler flags which lead to a large binary difference. We then use these compiler flags on the fuzzing benchmarks.

4 APPENDIX

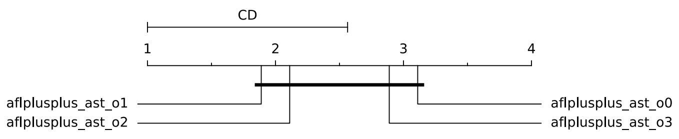

Fig. 1. Critical Difference Plot according to Demsar et al. [Demšar 2006], compares all benchmarks results with each other according to their average rank (in terms of edge coverage) and the statistical significance between them. Average rank is calculated based on medians of coverage on each benchmark, then averaged across all benchmarks. Fuzzers connected by the bold line indicate that there is no significant difference between them.

benchmark	flag	#nodes	#edges
aspell	-O0	243427	536133
aspell	-O1	165364	390152
aspell	-O2	174053	409345
aspell	-O3	180679	427503

Table 1. Comparing control flow graphs (binary based) of different -O levels. Results obtained with Angr [Shoshitaishvili et al. 2016].

covered_by	not_covered_by	#regions
-O0	-O1	212
-O0	-O2	115
-O0	-O3	120

Table 2. Incomplete table comparing unique regions covered by different -O levels on *bloaty fuzz target*.

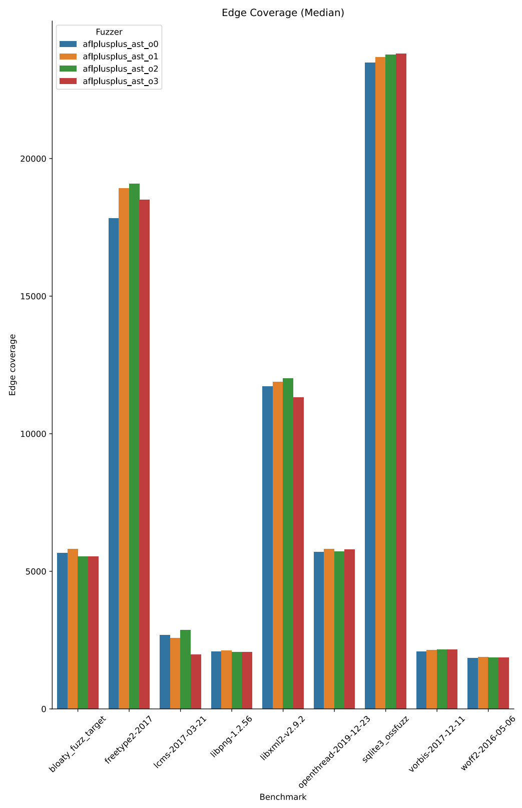

Fig. 2. Edge coverage across all preliminary benchmarks results (median 3 trials).

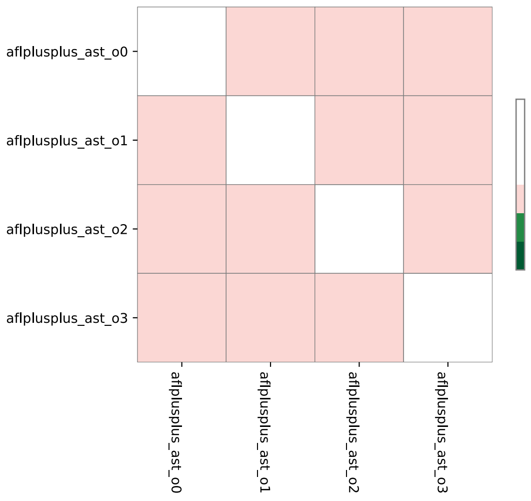

Fig. 3. Mann-Whitney test (Red marks a rejected null hypothesis) for vorbis benchmark.

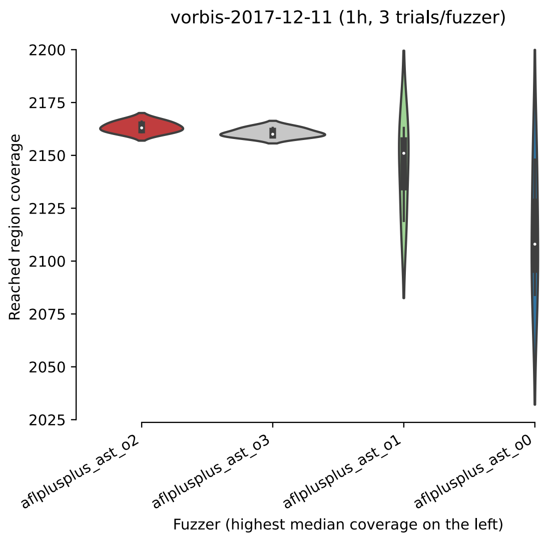

Fig. 4. Sample Violin plot for vorbis benchmark.

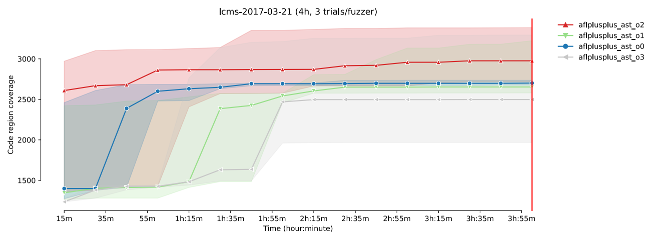

Fig. 5. Coverage growth across 4 hours of fuzzing.

REFERENCES

Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* 7, 1 (2006), 1–30. http://jmlr.org/papers/v7/demsar06a.html

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

LLVM. 2022. Clang documentation. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

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

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

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