LibAFL_libFuzzer: LibFuzzer on top of LibAFL

Addison Crump
CISPA
addison.crump@cispa.de

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Dominik Maier
Google Inc.
dmnk@google.com

Dongjia Zhang
The University of Tokyo
toka@aflplus.plus

***Abstract*—General-purpose fuzzing has come into the public eye, with many researchers developing new fuzzers to improve on the state of the art. LibAFL, developed by the group which originally made AFL++, offers researchers the ability to develop fuzzers at a component level, allowing researchers to simply develop their own components rather than modifying an existing fuzzer. This allows for more straightforward comparisons of fuzzers, allowing researchers to experiment with the removal and addition of individual components, without compromising on the flexibility of fuzzer development. To demonstrate this flexibility and offer alternative frontends to the community, we developed two fuzzers: LibAFL_libFuzzer and AFLrustrust, the former of which is discussed here as a drop-in replacement for LibFuzzer and the latter in a sister report as a drop-in replacement for AFL++. We find that LibAFL_libFuzzer performed very well on the coverage benchmarks while struggling with the bug-based benchmarks conducted in the SBFT fuzzing competition, and discover and analyse which fuzzer features and bugs led to this underperformance.**

I. Introduction

The Fuzzing community is very active and prolific, with an always growing number of proposed ideas and prototypes [1]. In practice, however, for generic fuzzing three main engines are widely used and are AFL++ [2], that is gradually replacing AFL [3], LibFuzzer [4] and HonggFuzz [5].

As a spin-off of AFL++, in the last two years we started developing a new fuzzing framework to cope with the extensibility problem of this widely used but monolithic fuzzer. This framework, LibAFL [6] is not a tool by itself but rather a collection of Rust libraries to write fuzzers. While power users appreciate the flexibility of writing custom fuzzers with LibAFL, other users with simpler needs are still recommended to use AFL++.

LibFuzzer [4], being integrated with LLVM, is a very commonly used fuzzer runtime. Recently, however, it has been put on maintenance mode [7]. To offer continued support and newly-discovered techniques to LibFuzzer-oriented fuzz harnesses, we developed LibAFL_libFuzzer, a near-complete replacement for LibFuzzer with support for the most common features without introducing new build requirements. At the time of the competition, it was still in a developmental state.

II. LibFuzzer on FuzzBench

We want to replicate the configuration of LibFuzzer that is used in FuzzBench to demonstrate the compatibility of LibAFL_libFuzzer with LibFuzzer harnesses. Specifically, LibAFL_libFuzzer requires no additional instrumentation over LibFuzzer and intentionally restricts itself to LibFuzzer compatible mutators and instrumentation; where LibFuzzer can be used, so too can LibAFL_libFuzzer.

To this end, we use not only the same instrumentation, but also the very same flags from the `libfuzzer` fuzzer and provide support for `LLVMFuzzerCustomMutator` and `LLVMFuzzerCustomCrossover`. With the fuzzer in full operation, the only observable distinction between LibFuzzer and LibAFL_libFuzzer to both the user and the system under test is the output format and improved performance.

III. Implementing LibAFL_libFuzzer

As described in the LibAFL paper [6], several components must be defined to build a fuzzer based on LibAFL.

A. Executor

LibAFL_libFuzzer, like LibFuzzer, will use in-process fuzzing and launch new jobs upon a crash or timeout. The jobs will be launched by a simple forking manager, which is the standard launcher for using LibAFL unparallelised. The executor will accept an input from the scheduler and execute it via a wrapper to `LLVMFuzzerTestOneInput`. The wrapper catches C++ exceptions to prevent unnecessary restarts when discovering exception-based crashes. Other than this, the executor is a very standard LibAFL-style executor.

B. Feedback

LibAFL_libFuzzer offers a large selection of feedback options for maximising compatibility. For SBFT, we will collect map coverage feedback and time feedback on all executions and comparison log feedback on the tracing stages (see III-D3 for details). These feedbacks will be used to determine interestingness and infer data to be used by mutators.

To determine if a fuzzer has identified a fuzzer objective, we utilise crash and timeout feedbacks to determine if the program has crashed or hung during execution. If the target does either, it is considered a fuzzing solution and added to the corpus of solutions.

C. Scheduler

To ensure inputs are selected to minimise time and maximise the potential of each input, we use an AFL-style "fast" power schedule to select high energy inputs. This schedule is wrapped with a scheduler which employs a corpus minimisation strategy to select inputs which cover the set of observed indices while minimising their execution time.

D. Stages

For SBFT, the fuzzer will leverage many different stages:

1) Generalization and Grimoire *Mutation:* Several fuzzbench targets are primarily structured text-based inputs. When provided with a corpus, we automatically determine if it is primarily text-based and, if so, will enable the Grimoire mutation strategy [8]. Otherwise, Grimoire-style mutations will be disabled and these stages will be skipped. This is discussed in further detail in III-E.

2) Calibration: To identify and prioritise inputs which are well-behaved and most interesting, we utilise a calibration stage which identifies coverage bitmap instability and typical execution times. This information is forwarded to the scheduler which will use it to prioritise stable, fast-executing inputs that maximise coverage map exploration.

3) Tracing: When comparison logging is enabled, the tracing stage will execute the target and collect comparisons performed by the target. This information will later be used to replace regions in the original input with the values they were detected as compared against. This is discussed in further detail in III-E.

4) Mutational: LibAFL_libFuzzer employs many mutational stages which are enabled based on the presence of custom mutators and custom crossover methods provided by the system under test. They are discussed in detail in the next section.

E. Mutator

In order to fully support targets which use LibFuzzer, we offer mutators to the system under test when they use their own mutations and when they do not. In this way, we maximise our compatibility.

1) Standard Mutation: With LibFuzzer, a system under test may provide the methods `LLVMFuzzerCustomMutator` and `LLVMFuzzerCustomCrossover` to provide custom mutations and custom crossover mutations, respectively [9]. When neither of these methods are detected, we utilise an AFL++-style scheduled mutator[1] [2] with LibAFL's AFL-style havoc mutations [6], including crossover[2].

2) Custom Mutation, No Crossover: When only `LLVMFuzzerCustomMutator` is detected, the mutator used only invokes the target-provided mutator. This mutator may internally refer to `LLVMFuzzerMutate` to access LibFuzzer's mutator. We instead provide an AFL++-style scheduled mutator [2] with LibAFL's AFL-style havoc mutations [6], but with crossover disabled as to not inject raw corpus entries into whatever the target may be mutating. Instead, the havoc crossover mutations are executed as a separate mutation stage, since we cannot guarantee the user intends to use bytes from raw corpus entries within their custom mutator.

3) Custom Crossover, No Mutation: When only `LLVMFuzzerCustomCrossover` is detected, the standard mutator is used with havoc mutations, excluding crossover. The custom crossover provided by the user is invoked in a separate pass, with the same standard mutator provided if it is invoked by the custom crossover mutator.

4) Custom Mutator and Crossover: If both `LLVMFuzzerCustomMutator` and `LLVMFuzzerCustomCrossover` are detected, the strategies mentioned in and are combined, deferring to the custom mutators provided by the user.

5) Input2State: When comparison logging is enabled, byte sequences in the inputs which were detected in failed comparisons by the tracing stage will be replaced with the intended comparison. Additionally, LibFuzzer's "interceptor" methods for common comparison methods (e.g., memcmp) are implemented to intercept and inspect comparison method parameters and results. This mutation allows us to overcome the issue of complex comparisons which prevent further exploration in the target. This mutator also conditionally uses the custom mutator.

6) Grimoire*:* When the target's inputs are detected to be primarily structured text, Grimoire-style mutations [8] will be used to selectively identify and replace tokens, byte sequences, and other text-based data within the input to maximise exploration of textual programs.

IV. Evaluation Results

In the SBFT final evaluation, LibAFL_libFuzzer performed reasonably well, but had notable issues on several targets. The results for both the bug[3] and coverage[4] are publicly available. These results speak largely for themselves in both rank and performance, so the discussion below deals primarily with identifying in what aspects LibAFL_libFuzzer must improve.

Our primary goal of outperforming LibFuzzer was largely achieved, excluding by rank in the bug-based experiment (in which LibAFL_libFuzzer ties with LibFuzzer). We discuss potential reasons for this unexpected underperformance in the next section.

[1] In the first round, we utilised MOpt, but switched to a standard scheduled mutator for the second round. It is unclear what performance impact this had on the overall performance at this time, and will likely become optional in a future iteration.

[2] This feature was in a bad state during the competition, as we did not expect these custom mutators to be present in targets for the competition. Though non-functional during the competition, we have planned dedicated support as custom mutations are used by OSS-Fuzz [10] and in "real" campaigns with LibFuzzer harnesses.

[3] https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Bug/index.html

[4] https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Coverage/index.html

A. Bug Benchmark Weaknesses

LibAFL_libFuzzer struggled in the bug benchmarks, finding (in most cases) all of the bugs discovered by others, excluding two benchmarks: `assimp_assimp_fuzzer_4d451f` and `file_magic_fuzzer_2d5f85`. In both of these benchmarks, LibAFL_libFuzzer underperformed due to a bug in the OOM handling, causing the fuzzer to terminate and not perform any further fuzzing. While disappointing, it is good that we can clearly point to what is failing and remedy this issue in the future (in this particular case, a two- or three-line change to how OOMs are detected is all that is necessary). We discovered these issues via logs provided by the competition organisers[5].

Excluding these two benchmarks, LibAFL_libFuzzer outperforms or has equal performance in overall bug discovery when compared to all other fuzzers undergoing trials. It performed roughly in the middle of the pack in terms of when these bugs were discovered, indicating that the corpus or mutation scheduling strategy used may not be optimal.

B. Coverage Benchmark Weaknesses

LibAFL_libFuzzer generally performed well in the coverage experiment, but had a low average normalised score. Notably, LibAFL_libFuzzer had the highest achieved median score (shared by hastefuzz and aflplusplusplus). This indicates the presence of one to many outliers, which we found to be:

1) `draco_draco_pc_decoder_fuzzer` – "Interceptor" functions did not work correctly, preventing the Input2State mutator (III-E5) from solving the header magic correctly.
2) `dropbear_fuzzer-postauth_nomaths` – Fuzzer harness contained a `LLVMFuzzerCustomMutator`, but the LibAFL_libFuzzer code which handled this was in a bad state at the time of competition (de-prioritised as it was not considered in-scope for the competition).
3) `proj4_proj_crs_to_crs_fuzzer` – Input shape for this target is two lines of floating point numbers; LibAFL_libFuzzer is generally optimised for byte manipulation rather than string manipulation, and generally had worse performance on this target.

The issues which caused the first two targets to underperform so dramatically are being resolved, and will be corrected by the time of the workshop. We are exploring options for string-specialised mutations, but how such mutators will be implemented is not known at the time of writing.

V. Summary

While LibAFL_libFuzzer does not propose new fuzzing capabilities or special features unavailable in other fuzzers, it does provide a fuzzer runtime which offers near-complete compatibility with LibFuzzer, one of the most widely used fuzzers. With developers able to easily switch out to a new option with near-complete compatibility, we provide support for most fuzzing configurations that rely on LibFuzzer, which has since entered maintenance mode. Additionally, by utilising LibAFL, we offer developers access to modern fuzzer algorithms without the need to dramatically change their build configurations or add new tools.

We are very pleased with the results, despite the setbacks that the fuzzer encountered, and look forward to improving the tool for future use in fuzzing campaigns as a LibFuzzer alternative.

VI. Acknowledgements

This work was supported by the European Research Council (ERC) under the consolidator grant RS^3 (101045669). Additionally, we would like to thank Dongge Liu and Jonathan Metzman especially for their support with Fuzzbench during the competition, as well as Marc Heuse ("van Hauser") for his insight on potential reasons for performance differences between fuzzers.

References

[1] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, nov 2021.

[2] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[3] M. Zalewski, "American Fuzzy Lop - Whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016, [Online; accessed 10 April. 2022].

[4] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, Sep. 2018, [Online; accessed 10 April. 2022].

[5] R. Swiecki, "Honggfuzz," https://github.com/google/honggfuzz, [Online; accessed 10 April. 2022].

[6] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22. ACM, November 2022.

[7] LLVM Project, "libFuzzer - Status," https://llvm.org/docs/LibFuzzer.html#id14, 2022, [Online; accessed 13 Jan. 2023].

[8] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing structure while fuzzing," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1985–2002. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[9] Google, "Structure-Aware Fuzzing," https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md#example-compression, [Online; accessed 20 Jan. 2023].

[10] "Google OSS-Fuzz: continuous fuzzing of open source software," https://github.com/google/oss-fuzz, [Online; accessed 10 April. 2022].

[5] https://storage.googleapis.com/fuzzbench-data/index.html?prefix=2023-03-06-sbft23-bug/experiment-folders/