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

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 in favour of another fuzzer, Centipede [7], which has different priorities from the original LibFuzzer [8]. To offer continued support and newly-discovered techniques to LibFuzzer-oriented fuzz harnesses, we plan to write LibAFL_libFuzzer, a near-complete replacement for LibFuzzer with support for the most common features without introducing new build requirements.

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 can LibAFL_libFuzzer.

To this end, we use not only the same instrumentation, but most of the flags from the `libfuzzer_fork_parallel` 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 low-level message passing server, which is the standard launcher for LibAFL. The executor will accept an input from the scheduler and execute it via `LLVMFuzzerTestOneInput`. 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-D2 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) 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.

2) 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.

3) 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 a MOPT mutator [10] with LIBAFL's AFL-style havoc mutations [6], including crossover.

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 a MOPT mutator [10] 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. 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.

IV. 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 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 excited to see LIBAFL_LIBFUZZER perform on FUZZBENCH and to provide developers with a LIBFUZZER alternative built on LIBAFL.

REFERENCES

[1] V. 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*, no. 01, oct 5555.

[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] Google, "Centipede," https://github.com/google/centipede, 2022, [Online; accessed 13 Jan. 2023].

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

[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] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu