AFLrustrust: A LibAFL-based AFL++ prototype

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Dominik Maier
Google Inc.
dmnk@google.com

Dongjia Zhang
The University of Tokyo
toka@aflplus.plus

Addison Crump
CISPA
addison.crump@cispa.de

***Abstract*—In this paper, we present a firts attempt in rewriting the widely used fuzzer AFL++ as a frontend of LibAFL, our new framework for fuzzers development. This proptype, AFLrusrust as it is written in the Rust programming language, was evaluated in the SBST'23 Fuzzing Competition with great results even being just a first attempt with missing components that are still under development.**

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 there are three main engines that are widely used. These are AFL++ [2], which is gradually replacing AFL [3], LibFuzzer [4] and HonggFuzz [5].

As a spin-off of AFL++, over the last two years, we developed 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, most users still prefer AFL++ for its out-of-the-box experience that fits most use-cases well.

To bridge the gap between our two projects, and solve the problem of adding additional fuzzing algorithms to AFL++, we plan to rewrite AFL++ as a frontend of LibAFL. While this process will take many months, we can already provide a LibAFL-based fuzzer that mimics AFL++, just without the many command line options the main project provides to customize the fuzzing behavior.

II. AFL++ on FuzzBench

Our goal is to faithfully replicate the configuration of AFL++ that is used in FuzzBench, as it is the best performer in a generic setup.

This configuration consists of the modified SanitizerCoverage `trace-pc-guard` module shipped in AFL++ as part of the LLVM-based instrumentation with `afl-clang-fast`. Here, the AFL++ edge coverage logging routine is inlined in the LLVM IR, while still using the guards generated with `trace-pc-guard`. This pass breaks the direct edges of the Control Flow Graph in each function, inserting an intermediate basic block and this allows to precisely count the edges. The coverage is non-colliding: AFL++ adapts the size of the shared coverage map to the number of instrumented edges, and thus it doesn't suffer from the well-known collision problem.

The other enabled option is the compilation of a secondary binary with the CmpLog instrumentation. The second binary logs the content of each comparison instruction and each routine with two pointers as arguments in a shared memory region, from which the fuzzer can read. AFL++ can use this runtime information to run an enhanced version of the RedQueen [7] mutator.

The last option, `dict2file`, extract the constants operators used in comparisons-related functions such as `strcpy` during compilation. These tokens are then used to build a dictionary for the fuzzer.

III. Implementing AFLrustrust

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

A. Executor

To emulate the AFL++ behaviour, the executor that must be used is a forkserver. The forkserver implementation in LibAFL supports the binaries compiled with `afl-clang-fast` from AFL++ and their advanced features such as shared memory input delivery and CmpLog instrumentation.

B. Feedback

The feedback used is the maximization of each entry of a coverage map. The coverage map is the one created by the AFL++ target binary. It is exposed to LibAFL with a shared-memory-based map observer. As objective, we simply consider every crashing input.

C. Mutator

The mutator is based on MOpt and schedule two sets of operations available in LibAFL, the bit-level havoc mutations and the token-based ones in order to use an user-supplied or autogenerated dictionary – with `dict2file` in this case.

D. Scheduler

The next input to fuzz is chosen by reusing the same algorithms of AFL++, reimplemented in LibAFL. Corpus culling is done by selecting a minimal set of testcases covering every edge seen so far with a weighted prioritization based on the testcase length and the execution time. The selection from this pool of testcases is then performed with the AFL++ weighted scheduler using the *explore* energy assignment scheme [8].

E. Stages

The stages that compose the fuzzer are four, starting with *calibration*, the stage used to measure stats about the current testcase such as stability and average execution time. Then, a tracing stage is used to run the target under a second forkserver executor with a CmpLog enabled binary to collect the cmp traces into the fuzzer metadata. After this one, and depending on it, there is the *input-to-state* stage that uses the CmpLog metadata to match tokens in the input with the various I2S mutators. In the end, a mutational stage with the classic havoc mutations is used and the energy is assigned using the power schedule from the metadata generated with the weighted scheduler.

IV. SBST'23 Competition Results

The SBST'23 fuzzing competition is composed of two experiments in which AFLrustrust was evaluated versus 11 fuzzers, AFL++ included.

The first experiment [1] is coverage-based in which the fuzzers are compared using the uncovered branch coverage over 23 hours on 38 different programs from OSS-Fuzz [9].

In terms of average rank, AFLrustrust placed 5th, and in terms of average normalized score[2] it took the 4th position. In both the ranking, it was really close to AFL++ which took a position in front of our tool in both the scores.

The second experiment [3] is bugs-based in which the fuzzers are evaluated in their ability to discover crashes that are linked to bugs in 15 vulnerable applications from OSS-Fuzz during 23 hours runs.

In terms of average rank, AFLrustrust wins the first position this time and the second place, with a tie in the score with the first fuzzer Pastis which is a hybrid fuzzer while our approach is purely based on coverage-guided fuzzing, in the average normalized score [4] ranking.

While the performance on the bugs-based dataset seems great, we believe that our tool was penalized in the coverage-based experiment due to the score 0 assigned to the `zlib_zlib_uncompress_fuzzer` benchmark. For an unknown reason, the fuzzer failed to start the experiment with this target program but we could not replicate the failure locally as AFLrustrust builds and runs fine this zlib fuzzer.

V. Discussion

The proposed fuzzer based on LibAFL is a first attempt at the upcoming rewriting of AFL++ as a frontend of LibAFL and thus may be incomplete or even with buggy components. In security research, often the best fuzzer is not the one that finds slightly more coverage than the others but the one that fits the user needs, and LibAFL aims at this. On the other hand, beginners and developers need an off-the-shelf environment that can fuzz a target with a minimal setup, as AFL does, and an AFL++ clone based on LibAFL can provide the best of both worlds.

The results of the fuzzing competition shows that AFLrustrust shines even if it is just a prototype. The lack of uncovered coverage compared to the original tool, AFL++, is due to the missing implementation of the input-to-state mutator in the same advanced way as AFL++ which includes advancements from RedQueen [7], Weizz [10] and others original algorithms to approximately solve branch constraints developed during the years. This is now a work in progress in LibAFL and an equivalent mutator are under development and in the roadmap of the rewrite of AFL++ on top of LibAFL.

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

[2] This score is based on the average of per-benchmark scores, where the score represents the percentage of the highest reached median code coverage on a given benchmark.

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

[4] This ranking is based on the average of per-benchmark scores, where the score represents the percentage of the highest reached median bug coverage on a given benchmark.

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 15 March. 2023].

[4] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, Sep. 2018, [Online; accessed 15 March. 2023].

[5] R. Swiecki, "Honggfuzz," https://github.com/google/honggfuzz, [Online; accessed 15 March. 2023].

[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] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[9] "Google OSS-Fuzz: continuous fuzzing of open source software," https://github.com/google/oss-fuzz, [Online; accessed 15 March. 2023].

[10] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397372