USENIX Security '25 Artifact Appendix
ELFUZZ: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space

Chuyang Chen
The Ohio State University
chen.13875@osu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

A Artifact Appendix

This artifact appendix is a self-contained document that describes the roadmap for the evaluation of the artifacts of the paper "ELFUZZ: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space." It includes the hardware, software, and configuration requirements for the evaluation, as well as the process for reproducing the results and claims presented in the paper.

The artifacts have acquired the "artifact available" and "artifact functional" badges after the artifact evaluation. The full replication experiments for the "artifact reproduced" badge were not finished within the one-month time span of the artifact evaluation process due to the requirements of a significant amount of time and substantial computational resources. However, our testing shows that the artifacts function well to reproduce all the results in the paper with enough time and computational resources provided. Furthermore, we will continue to improve the artifacts for better usability and bug fixes after the artifact evaluation.

A.1 Abstract

The artifacts contain the implementation of ELFUZZ, the source code of the benchmarks and baselines compared with ELFUZZ in the evaluation, the evaluation results, and the data required to reproduce the evaluation. It also includes a Docker image that preserves the environment and setups of the evaluation to facilitate one-touch replication.

A.2 Description & Requirements

This section outlines the hardware and software requirements necessary to recreate the experiment environment and setups described in the paper, as well as the steps to acquire the benchmarks and baselines used in the evaluation.

A.2.1 Security, privacy, and ethical concerns

The artifact does not perform any operations that may compromise the security or privacy of the evaluators who execute it.

A.2.2 How to access

The replication package, containing the code and data, is published on Zenodo. The download link of the version at the time of artifact evaluation is https://doi.org/10.5281/zenodo.16741080. Updated versions will be hosted at https://doi.org/10.5281/zenodo.15833146. The source code of ELFUZZ and the replication package is developed in the GitHub repository at https://github.com/OSUSecLab/elfuzz.

A.2.3 Hardware dependencies

The experiments require a GPU with 26 GiB VRAM and CUDA support to run a local CodeLlama-13B model. The intermediate data and results produced or downloaded by the experiments will occupy approximately 100 GiB of disk space. In our evaluation, we use 30 processes to accelerate the experiments. It is recommended that the evaluators' machines be equipped with CPUs with at least this number of cores. Fewer CPU cores do not affect the results of the experiments, but a longer time to finish the experiments should be expected. The specific GPU and CPUs used in the original evaluation are one NVIDIA H100 Tensor Core GPU and two AMD EPYC 7251 CPUs.

A.2.4 Software dependencies

The experiments do not depend on a specific operating system. A Docker installation that is compatible with version 24.0.7 is required. The host machine should also have the CUDA toolkit installed.

A.2.5 Benchmarks

Benchmarks used in the evaluation will be automatically cloned from their official Git repositories or downloaded from the official sites during the experiments. Dockerfiles from the Fuzzbench project[1] and the OSS-Fuzz project[2] are modified for replicable building environments for the six benchmarks other than `cvc5`. The Dockerfile for `cvc5` is implemented from scratch. All Dockerfiles are included in the source code tarball of the artifacts.

[1] https://github.com/google/fuzzbench
[2] https://github.com/google/oss-fuzz

A.3 Setup

This section describes how to set up and configure the environment to replicate the evaluation. It also includes instructions on the functionality validation of the replication package.

A.3.1 Installation

First, follow the official instructions[3] to install Docker. Then, set up the core pattern in the host machine as required by AFL++ later. Hereafter, "$" indicates user inputs, and ">" indicates program outputs.

```
$ echo core > /proc/sys/kernel/core_pattern
> ...
```

Now, import the Docker image and launch the container:

```
$ zstd -d elfuzz_docker_<timetag>.tar.zst
$ docker load --input elfuzz_docker_<timetag>.tar
$ mkdir -p /tmp/host
$ docker run --storage-opt size=100G \
             --cpus 30 \
             --add-host=host.docker.internal:host-gateway \
             -v /tmp/host:/tmp/host \
             -v /var/run/docker.sock:/var/run/docker.sock \
             --name elfuzz \
             -it ghcr.io/osuseclab/elfuzz:25.07.2
```

The commands should lead you into the container where the experiments will happen.

A.3.2 Initilizing the environment

After entering the Docker container, run the following commands to enable sibling containers.

```
$ sudo chown -R appuser:appuser /tmp/host/
$ elfuzz setup
> ...
$ exit
```

The two commands will exit the container. Now, restart the container for the settings to take effect.

```
$ start -ai elfuzz
```

Then, the following command will download all the data files from Zenodo and place them in the correct locations.

```
$ elfuzz download
```

You also need to configure your Hugging Face token:

```
$ elfuzz config --set tgi.huggingface_token <YOUR_TOKEN>
```

A.3.3 Functionality Validation

After initializing the environment, you can run mini-versions of the experiments presented in the paper to validate that all of them function well. These commands adopt the same settings as the original experiments, but decrease the parameters such as time limits and the number of evolution iterations for quick validation. Each command can take 5 minutes to 2 hours to complete. §A.4 will give detailed explanations of these commands.

```
$ elfuzz synth -T fuzzer.elfuzz \
               --use-small-model \
               --evolution-iterations 3 \
               jsoncpp
$ elfuzz synth -T grammar.glade re2
$ elfuzz synth -T semantics.islearn cvc5
$ elfuzz produce --time 10 -T glade jsoncpp
$ elfuzz minimize -T glade jsoncpp
$ elfuzz run rq1.seed_cov -T glade jsoncpp
$ elfuzz run rq1.afl --fuzzers glade \
                     --repeat 1 \
                     --time 60 \
                     jsoncpp
$ elfuzz run rq2.afl --fuzzers glade \
                     --repeat 1 \
                     --time 60 \
                     jsoncpp
$ elfuzz run rq2.triage
$ elfuzz run rq2.real_world --time 60
$ elfuzz run rq3
```

These commands should output messages like "ELFuzz fuzzers successfully synthesized" without being interrupted by errors.

[3] https://docs.docker.com/engine/install/

A.4 Evaluation Workflow

This section outlines the major claims presented in the evaluation part of the paper (§A.4.1). It describes how to conduct the experiments to fully reproduce the results presented in the paper, thereby supporting these claims (§A.4.2). However, these experiments require a significant amount of time and substantial computational resources. Thus, another option is to run the smaller-scale experiments (§A.4.2), which share the same settings as the full-scale experiments but are conducted for a shorter time or with fewer repetitions. Results produced by the small-scale experiments should also support our claims, although not 100% replications of those in our original paper. We will explain how to conduct them and draw the same claims from the results.

A.4.1 Major claims

The major claims of the paper are as follows:

- **C1.** Fuzzers synthesized by ELFUZZ significantly outperform the state-of-the-art generation-based fuzzers, respecting the coverage of the produced test cases and the promotion that the test cases bring to later mutation-based fuzzing processes. This is proven by the input generation experiments (E1) and the mutation-based fuzzing experiment (E2) in RQ1.
- **C2.** Fuzzers synthesized by ELFUZZ significantly outperform the state-of-the-art generation-based fuzzers when being used to find artificially injected bugs, and they can find previously unknown bugs in real-world SUTs. This is proved by the bug-finding experiments on bug-injected benchmarks (E3) and the real-world bug-finding experiment on `cvc5` (E4).
- **C3.** Fuzzer space contributes the most to the performance of ELFUZZ among all the components. This is proved by the input generation experiments (E5) and the LLM-driven evolution processes (E6).
- **C4.** Fuzzers synthesized by ELFUZZ are interpretable and extensible. This is proved by case studies on the fuzzer code (E7) and manual adaptation of ZEST, another input-generation technique, onto the fuzzers (E8).

A.4.2 Full-scale experiments

Below are instructions on conducting full-scale experiments to replicate the results in the paper. It also lists the expected outputs and how they support our claims. Note that if you have validated the functionality of the replication package

before the experiments, please reset the Docker container (remove, re-launch, and re-initialize it) to avoid data pollution caused by the results produced by the functionality validation process.

- **E1, E5, and E6.** [1 human-hour + 10 compute-day + 50GiB disk] Inside the Docker container, use the following command to synthesize the fuzzers using ELFUZZ or its variants (E6), or mine grammars and semantic constraints using the baselines:

  ```
  $ elfuzz synth -T <baseline> <benchmark>
  ```

  Please refer to the help message (`--help`) on what values `<baseline>` and `benchmark` can take. The following commands conduct the input generation experiments (E1 and E5):

  ```
  $ elfuzz produce -T <baseline> <benchmark>
  $ elfuzz minimize -T <baseline> <benchmark>
  $ elfuzz run rq1.seed_cov -T <baseline> <benchmark>
  $ elfuzz run rq3
  ```

  The coverage of the generated seed test cases will be recorded in an Excel sheet, which can be viewed via

  ```
  $ pyexcel view /elfuzz/analysis/rq1/results/seed_cov.xlsx
  ```

  The coverage of seed test cases generated by ELFUZZ is expected to be significantly higher than that generated by other methods (E1 results for C1). Similarly, you can view the coverage of seed test cases generated by ELFUZZ variants via

  ```
  $ pyexcel view /elfuzz/analysis/rq3/rq3_ablation.xlsx
  ```

  The coverage of seed test cases generated by ELFUZZ-NOSP, ELFUZZ-NOCP, and ELFUZZ-NOIN will show slight decreases, while that generated by ELFUZZ-NOFS will show significant decreases (E5 results for C3). The coverage trends of candidate fuzzers of the four variants during evolution can be viewed via

  ```
  $ pyexcel view /elfuzz/analysis/rq3/rq3_evolve_cov.xlsx
  ```

  The values will display curves similar to those shown in Figure 12 of the paper if being drawn (E6 results for C3).

- **E2 and E3.** [1 human-hour + 20 compute-day + 5GiB disk] The following command runs the mutation-based fuzzing experiments in RQ1 (E2) and collects and analyzes the results:

  ```
  $ elfuzz run rq1.afl --fuzzers <baseline_list> \
                       --repeat 10 <benchmark_list>
  ```

  The following commands will run the mutation-based fuzzing campaign in RQ2 (E3) and collect and analyze the results:

  ```
  $ elfuzz run rq2.afl --fuzzers <baseline_list> \
                       --repeat 10 <benchmark_list>
  $ elfuzz run rq2.triage
  ```

  You can view the averaged coverage trends during AFL++ fuzzing campaigns for RQ1 and trends of triggered bugs via

  ```
  $ pyexcel view /elfuzz/rq1/results/rq1_sum.xlsx
  $ pyexcel view /elfuzz/rq2/results/rq2_count_bugs.xlsx
  ```

  The data will show curves similar to those in Figures 8 and 9, i.e., the curves of ELFUZZ will be higher than that of others (E2 for C1 and E3 for C2).

- **E4.** [8 human-hour + 14 compute-day] The following command runs the real-world bug-finding experiment:

  ```
  $ elfuzz run rq2.real_world
  ```

  The triage and analysis of the results can only be done manually. The data tarball contains the bug-triggering test cases we found, though, in `rq2/results/cvc5_bugs`. You can check whether they actually trigger the bugs of the corresponding versions of `cvc5` (E4 for C2).

- **E7 and E8.** These two empirical case studies were done manually. The README file in the Docker image contains suggestions on how to replicate them (E7 and E8 for C4).

- **Reproducing the figures and tables.** Use the following command to reproduce the figures and plots presented in the paper:

  ```
  $ elfuzz plot --all
  ```

  Note that we cannot include proprietary fonts in the replication package. The appearance of the final rendered figures may be different from that in the original paper.

Please refer to the README files in the Zenodo repository, source code tarball, and Docker image tarball for more details.

A.4.3 Small-scale experiments

The full-scale experiments require a significant amount of time and substantial computational resources. Instead, you can choose to run some experiments at smaller scales, and the results can also support our claims. Below are instructions.

- **E1 and E5.** [1 human-hour + 1 compute-day + 25GiB disk] Unfortunately, decreasing the number of evolution iterations will significantly affect the performance of the synthesized fuzzers, which is expected. Thus, we cannot run the small-scale versions of the synthesis processes and E6. Therefore, we will utilize the original fuzzers included in the replication package. For E1 and E5, you can use the `--time 60` option for the `produce` subcommand to run the seed test case generation for 1 minute instead of 10. Even this short time limit should already demonstrate the advantage of ELFUZZ (C1) and the huge impact of excluding the fuzzer space from ELFUZZ (C3).

- **E2 and E3.** [1 human-hour + 5 compute-day + 5GiB disk] Note that using seed test cases generated in a time shorter than 10 min can weaken the advantage of ELFUZZ in AFL++ fuzzing campaigns, which is expected. Please reset the Docker container if you have run the above small-scale experiments before starting these two experiments, to ensure that you use the original test cases (generated in 10 minutes) included in the replication package. Then, you can change the value of the `--repeat` option from 10 to 3 to decrease the number of repetitions. Use the `--time 3600` option to run the AFL++ fuzzing campaigns for 1 hour instead of 24 hours. ELFUZZ should be able to show advantages within the first hour of fuzzing.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.