USENIX'23 Artifact Appendix: Systematic Assessment of Fuzzers using Mutation Analysis

Philipp Görz[1], Björn Mathis[1], Keno Hassler[1], Emre Güler[2], Thorsten Holz[1], Andreas Zeller[1], and Rahul Gopinath[3]

[1]CISPA Helmholtz Center for Information Security, Germany
[2]Ruhr-Universität Bochum, Germany
[3]University of Sydney, Australia

A Artifact Appendix

A.1 Abstract

We provide the source code of our benchmarking framework, which is written in Python. This includes the code to create the plots, written in Python and R. Additionally, we provide the seed corpus used as the initial input for the tool. Furthermore, we provide the artifacts from the evaluation stages, including the final databases from which the plots in the paper are generated.

Additionally, we provide notes and tooling for the two manual analyses performed for the paper.

A.2 Description & Requirements

Recreating the experimental setup requires a Linux system with docker installed. Additionally, the user needs to be in the docker group. To manage the python version and dependencies hatch is used. For details see the readme of the framework.

The evaluation requires a initial minimal seed corpus, which is provided by the zip file in the Zenodo link, it is the 'seeds/minimal' directory. Usage of this seed corpus is described in the readme of the framework.

Additionally, an environment setup script is provided by the framework, again see the readme for details.

This should (hopefully) be all that is required to recreate the experimental setup, while we have tested using the framework on some systems we can obviously not guarantee that it will work on all systems. If there are any issues please contact us.

The minimal hardware requirements are at least 16GB of RAM and 50GB of disk space. The RAM requirements scale with number of running instances, which are more likely limited by the number of cores available. The framework is designed to scale to with the number of cores.

For reference, the evaluation for the paper used four servers with Intel Xeon Gold 6230R CPUs, each with 52 cores and 188 GB RAM.

A.2.1 Security, privacy, and ethical concerns

The framework requires a user that is in the 'docker' group, this should be seen as equivalent to root access, although this way, we avoid running the whole framework as root. Furthermore, the provided setup script will disable ASLR on the system to stabilize the fuzzing results, but it also facilitates exploitation of security vulnerabilities. If this is a concern, comment out the respective line. The docker containers will access '/dev/shm' and '<project root>/tmp' on the host system, this is required for the shared memory and exchanging files. Other than for building the Docker images, no internet access is required.

A.2.2 How to access

The artifact consists of two parts: the main framework and the other artifacts, both are required to reproduce the results. From the 'Other Artifacts', only the seed corpus in the `mua-fuzzer-bench-eval-data.7z` archive under the directory 'seeds/minimal' is strictly needed to reproduce the results. The remaining files can be referenced to support the evaluation process as they contain our intermediate and final results.

'Framework - Source Code': `https://github.com/CISPA-SysSec/mua_fuzzer_bench/tree/b3cc3815f9dce9371eb5d461bb5beb888c032327`

'Other Artifacts': `https://zenodo.org/record/8060560`

A.2.3 Hardware dependencies

The evaluation framework runs on commodity hardware, but reproducing every result in the paper will consume a considerable amount of CPU ressources. For reference, the evaluation

for the paper used four servers with Intel Xeon Gold 6230R CPUs, each with 52 cores and 188 GB RAM. Thanks to the modular design, it is also possible to run the evaluation on a subset of fuzzers or programs.

A.2.4 Software dependencies

The evaluation framework depends on Linux with docker and hatch installed. The user needs to be in the docker group. We tested the framework on Ubuntu and Debian, but it should run on any distribution.

A.2.5 Benchmarks

The evaluation requires a initial minimal seed corpus, which is provided by the zip file in the Zenodo link, it is the 'seeds/minimal' directory. To reproduce the exact figures shown in the paper, we provide the result databases.

A.3 Set-up

See the 'Usage' section of the readme in the framework repository.

A.3.1 Installation

See the 'Installation' section of the readme in the framework repository.

A.3.2 Basic Test

See the 'Usage' section of the readme in the framework repository. For a basic test the `-fuzz-time` parameters of the commands shown can be reduced to one minute, `--instances` can also be reduced. Additionally, the command run under 'Basic Evaluation' can be manually aborted early via a keyboard interrupt (`Ctrl+C`) to reduce the number of evaluated supermutants. The commands under 'ASan' and '24 Hours' will only evaluate supermutants that have been tried for the 'Basic Evaluation'.

Expected output for the `coverage_fuzzing` command is the directory containing the coverage seed corpus (see the `-result-dir` argument), for the `eval` command the expected output are the databases placed at the path given by the `-result-path` argument.

A.4 Evaluation workflow

A.4.1 Major Claims

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Enumerate here the major claims (Cx) made in your paper. Follows an example:

(C1) : Computational effort is reduced by (on average) factor 3.8 by using supermutants. This is a side result from experiment (E1) described in section 5.2 and reported in Table 4.

(C2) : Different fuzzers show quite similar results. This is also proven by experiment (E1) described in section 5.2; the results are reported in Table 5 and illustrated in a Venn diagram (Figure 3).

(C3) : Coverage accounts for most mutants detected (97.5%) in our evaluation. This is the share of all mutants that were killed by the ensemble of all fuzzers during coverage fuzzing, as explained in Section 5.2 (paragraph *Results*). This number can be calculated from the data in Table 5, which is generated in experiment (E1).

(C4) : ASan moderately increases the number of killed mutants. In Section 5.3, we calculate this number per evaluated fuzzer. This is based on comparing the results from experiment (E2) shown in Table 6 with the results from experiment (E1) shown in Table 5.

(C5) : One hour of fuzzing after the seed coverage stage is sufficient to evaluate a supermutant. In experiment (E3), we re-run a random subsample for 24 hours and see that almost no additional mutants are killed (Table 7). This is described in Section 5.2.

(C6) : Most of the remaining mutants (84%) introduce a semantic change (*theoretically* detectable with a perfect oracle). This is based on manual analysis of non-killed mutants (E4). The result is described in Section 5.2.1.

(C7) : Mutations induced by our mutation operators are coupled to real faults, since 71% of the studied recent vulnerabilities in experiment (E5) can be re-introduced with our mutation operators. We explain this result in Section 5.4.

A.4.2 Experiments

(E1): [Basic Experiment] [16.36 CPU core years] The initial experiment as described in Section 5.2. Includes Phase I and Phase II.
How to: All setup and preparation is explained in the readme of the framework. Everything should be explained when following the instructions up to 'Basic Evaluation'.
Note that the `--seed-dir` should point to the extracted content of the `seeds/minimal` directory of the eval data archive.
Results: The resulting plots for experiments E1 to E3 can be are produced as described in the readme of the framework in the section 'Getting the Results'.

(E2): [ASan Experiment] [15.16 CPU core years] This experiment depends on E1, keep following the process as described in the readme of the framework to the section 'ASan'.

(E3): [24 Hours Experiment] [7.42 CPU core years] This experiment depends on E2, keep following the process as described in the readme of the framework to the section '24 Hours'.
Note that either the rerun json file can be adapted to contain 100 mutations or a manual interruption can be done once the 100 mutations are reached.
Now, finally, the results can be produced as described in the readme of the framework in the section 'Getting the Results'.

(E4): [Manual Analysis of Mutations] [8 human hours]: This is a manual analysis of mutations that are not killed to see if the created mutations are useful to evaluate fuzzers.
How to: Examine covered mutations that are not killed even after the 24 hour experiment, see Section 5.2.1. Just for reference, we provide our notes of the manual analysis in the Zenodo repository, under the `not_killed_24.xlsx` file.
Preparation: This experiment depends on the result of the previous experiment (E3), the following SQL query should be run on the database produced. Note that the `prepare_db` command needs to be run on the database, see 'Getting the Results' in the readme of the framework. The list of mutants that are still not killed after 24 hours is obtained with the following SQL query:

```
select completed_runs.prog, completed_runs.
    mut_id, directory, file_path, line,
    column, instr, funname, pattern_name,
    description, procedure from
    completed_runs
join mutations on mutations.prog =
    completed_runs.prog and mutations.
    mutation_id = completed_runs.mut_id
join mutation_types using (mut_type)
where num_confirmed == 0
order by random()
limit 120;
```

Note that some of the mutants can be in system libraries, which we have skipped during our manual analysis, this is also the reason why the limit is set to 120 instead of 100.
Note that the source code of the programs can be found under the `<project root>/tmp/programs` directory.
Execution: This is a manual experiment, where for each mutation the corresponding line and surrounding code is examined to decide if the mutation does not cause a semantic change, or if it does, whether it is detectable when using ASan or a simple crash oracle.

(E5): [Manual Analysis of Vulnerability Coupling] [8 human hours] This is a manual analysis to see if the mutations that are created simulate real vulnerabilities. This is described in Section 5.4.
How to: We regard a vulnerability to be reintroduced if the mutation causes the patched program to reintroduce the vulnerability. We provide the list of CVEs we analyzed in the Zenodo repository, under the `CVEs.xlsx`. The list of CVEs that we analyzed was obtained using the code in the file `cve-script.7z`, which uses the official CVE list as source. The script is written in Rust, though we would recommend to just re-examine the CVEs we analyzed.
Preparation: Required is the list of CVEs to analyze and a description of the mutation operators. Which can be found in the framework repository under `<project root>/mutation_doc.json`.
Execution: For each CVE, examine the patch if it would be introduced by a mutation operator or a combination of mutation operators. If so, the CVE is regarded as reintroduced.

A.5 Notes on Reusability

The framework is modular and allows to run on specified sets of fuzzers and programs for a chosen time. For details, consult the readme and help for the evalutation script (accessible with `-h`). The readme of the framework repository contains a section 'Extending the Tool', describing how the tool can be used to evaluate on other programs, fuzzers, and mutations.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.