This document is the introduction for our submission to the Artifact Evaluation of USENIX Security'22 for our paper titled "FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing".

File Structure

The submitted package consists of 5 folders.
1. **programs.** This folder stores the setups for bug injection on the 10 benchmark programs used in the paper evaluation. The usage of each folder for each program will be explained later. The injection results of the 10 programs are provided in the Section *FuzzBench Experiment*.
2. **FixReverter.** This folder stores the tools for bug injection**.**
3. **experiment-data.** This folder stores the fuzzbench experimental results used in the evaluation, for reproducing triage. It needs to be downloaded and extracted from https://drive.google.com/file/d/13gFuPpwRdhawkL4B9JG4IzBdCi6VkPLu/view?usp=sharing
4. **triage**. This folder stores the triage intermediate products, to reproduce Table 2, Figure 9 and 10 in the paper.
5. **scripts**. This folder stores scripts for reproducing results in the paper. Currently it contains only 1 script.

Main Workflow

The general workflow of the artifact has 3 steps. Firstly, we inject bugs into source code of target programs with FixReverter. Secondly, We run fuzzbench local experiments on the injected programs. Lastly, we triage injected bugs based on the fuzzbench experimental results.
The intermediate results of all steps are provided in the artifact, so you can skip any step of any program when following the workflow.

System Requirements

We ran the evaluation on the Ubuntu system. The hardware specs we used can be found in Section 5 of the paper. RAM of less than 200GB may fail the bug injection of some programs. We recommend CPUs with more than 24 cores, to speed up triage.

FixReverter Bug Injection

Next are the steps for injecting a program with FixReverter. The instructions are based on program lcms and similar steps apply to all 10 programs. For detailed instruction on injecting the other 9 programs, refer to programs/[PROGRAM_NAME]/README.pdf.

Tool Setup

Before we start
1. All tools in the FixReverter/clangTools folder are packed in a docker image with FixReverter/Dockerfile. To build the image and name it as "fr_clang", run command
   `cd FixReverter`
   `docker build -t fr_clang .`
   `cd ..`
   This process could take several hours, as it builds boost and llvm from source.
2. All tools in FixReverter/phasar_plugin folder are packed in a docker image with FixReverter/phasar_plugin/Dockerfile. To build the image and name it as "fr_phasar", run command
   `cd FixReverter/phasar_plugin`
   `docker build -t fr_phasar .`
   `cd -`
3. Running the semantic matcher of FixReverter resulted in out-of-memory in the evaluation. Therefore, we compromised on the soundness of the analysis for the scalability, and ported the semantic matcher to an older version of Phasar. To build the Docker image and name is as "fr_phasar_port", run command
   `cd FixReverter/phasar_plugin_port`
   `docker build -t fr_phasar_port .`
   `cd -`

Run Syntax Matcher

1. After the FR_clang image is ready, get into the clang directory of program lcms, by
   `cd programs/lcms/clang`
2. Create a docker container, by
   `docker run -ti --name=fr_lcms_clang -v $PWD:/src fr_clang bash`
   The -v options mounts [path/to/FixReverter]/programs/lcms/clang directory on the host machine onto the /src directory in the container.
3. Now you are at the /src directory of the docker container with a terminal. Setup lcms with
   `bash setup.sh`
4. Then build lcms with bear to generate the compilation database
   `bash build.sh`
5. Run the syntax matcher with
   `python3 /fixreverter/FixReverter/drivers/inject/driver.py -p`
   There will be an `apm.json` file storing the syntax match results in /src/tmp(mounted on programs/lcms/clang/tmp).
   Detach from the clang tools container with CTRL+p + CTRL+q

Run Semantic Matcher

1. Go to the semantic matcher folder by

cd ../phasar

And copy apm.json file, which is the input for the semantic matcher

cp ../clang/tmp/apm.json .

2. Start the docker container for the semantic matcher. Similarly, [path/to/FixReverter]/programs/lcms/clang directory on the host machine is mounted on the /src directory in the container.

docker run -ti --name=fr_lcms_phasar -v $PWD:/src fr_phasar bash

3. Download and build program for the semantic matcher

bash build.sh

4. Run the semantic matcher

bash run_phasar.sh

For some programs this step may take up to 200GB memory. The process will end with an error when it runs out of memory. It can take up to 3 days.

After the process finishes, the output file is stored at /src/out/dda.json.

Now we can stop the fr_lcms_phasar container with CTRL+D

Run Injector and Naive Bug Filter

1. Go back to the clang folder

cd ../clang

And copy the semantic matcher output from the previous step, or the provided one

cp ../phasar/out/dda.json .

Or

cp ../inject_products/dda.json .

Then go back to the fr_lcms_clang container

docker attach fr_lcms_clang

2. Move the semantic matcher output to the correct location.

mv ./dda.json ./tmp

This step is done in the docker container to avoid permission issues.

3. Clean the previous build

cd Little-CMS && make distclean && cd ..

4. Rewrite the program

python3 /fixreverter/FixReverter/drivers/inject/driver.py -i

5. Build the coverage binary

bash build_cov.sh

Errors are expected on this step.

6. Run NaiveBugFilter and re-inject with filtered bugs

python3 /fixreverter/FixReverter/drivers/inject/driver.py -r

7. Get the diff for the injection by

cd Little-CMS

make distclean

git diff >> fr_injection.patch

Now the lcms program is ready for fuzzing. See the next section, *Run FuzzBench Experiment* on how to fuzz it.

8. All 3 intermediate products, the apm.json, dda.json and inject.json are provided in programs/lcms/inject_products.

Count Patterns

1. To count patterns in injection stages and generate table 1 in the paper, use the following command:
   python3 artifact/scripts/count_pattern.py -y [path/to/apm.json] -m [path/to/dda.json] -f [path/to/inject.json]

FuzzBench Experiment

FuzzBench Setup

The integration with FuzzBench and bug triage scripts are uploaded on
https://github.com/SlaterLatiao/RevBugBench.

1. Before running the triage scripts, we need to set up FuzzBench. The version used by FixReverter is 65297c4c76e63cbe4025f1ce7abc1e89b7a1566c. The following commands will checkout the commit:
   git clone https://github.com/google/fuzzbench.git
   cd fuzzbench
   git checkout 65297c4c76e63cbe4025f1ce7abc1e89b7a1566c
   git submodule update --init
   cd ..

2. Set up FuzzBench prerequites as described in
   https://google.github.io/fuzzbench/getting-started/prerequisites/.

3. Clone RevBugBench repository
   git clone https://github.com/SlaterLatiao/RevBugBench.git

4. Copy benchmark programs of RevBugBench into FuzzBench
   cp -r RevBugBench/benchmarks/* fuzzbench/benchmarks
   If you want to use the programs you previously injected with clangTools, just replace the fr_injection.patch file with the one you generated on Step 7 of Run Injector and Naive Bug Filter.
   rm fuzzbench/benchmarks/lcms/fr_injection.patch
   cp programs/lcms/clang/Little-CMS/fr_injection.patch fuzzbench/benchmarks/lcms/
   Apply modifications of FuzzBench by RevBugBench
   cd fuzzbench
   git apply ../RevBugBench/fuzzbench/revbugbench.patch
   cd ..

5. Set up a config file following
   https://google.github.io/fuzzbench/running-a-local-experiment. An example can be found at

. Save the config file to fuzzbench/experiment-config.yaml.
We recommend at least 1 hour timeout in order to get some fuzzing results. The paper's evaluation used a timeout of 24 hours and 3 trials, on 5 fuzzers and 10 benchmark programs. The experimental results are provided in the experiment-data folder.

Run Experiment

1. Run the FuzzBench experiment with
```
cd fuzzbench
source .venv/bin/activate
PYTHONPATH=. python3 experiment/run_experiment.py \
--experiment-config experiment-config.yaml \
--benchmarks [list of program names] \
--experiment-name [an experiment name] \
--fuzzers [list of fuzzer names] \
-a
```

For example, fuzzing program lcms with the 5 fuzzers evaluated in the paper will be
```
PYTHONPATH=. python3 experiment/run_experiment.py \
--experiment-config experiment-config.yaml \
--benchmarks lcms \
--experiment-name exp \
--fuzzers afl aflplusplus eclipser fairfuzz libfuzzer \
-a
```

The fuzzers used in the evaluation are afl, aflplusplus, eclipser, fairfuzz and libfuzzer. The benchmark programs used in the evaluation are listed in
https://github.com/SlaterLatiao/RevBugBench/tree/main/benchmarks.
So for the full scale evaluation, the command will be
```
PYTHONPATH=. python3 experiment/run_experiment.py \
--experiment-config experiment-config.yaml \
--benchmarks binutils-fuzz_cxxfilt binutils-fuzz_disassemble curl lcms libpcap
libxml2_reader libxml2_xml proj4 usrsctp zstd \
--experiment-name exp \
--fuzzers afl aflplusplus eclipser fairfuzz libfuzzer \
-a
```
2. Note that the full scale evaluation takes 3*5*10=150 CPUs. If the machine does not have enough CPUs, consider dividing the programs into separate experiments.
As fuzzing is a random process, in order to reproduce the evaluation results (Table 2, Figure 9 and 10), the provided FuzzBench experimental results have to be used.
3. The *experiment-data* folder contains the experimental results used in the evaluation. We only uploaded files that are related to the evaluation to save space.

FixReverter Bug Triage

Setup

1. The triage takes a config file, config.ini. In this file 3 paths need to be set.
   - *workDir* points to a path where intermediate results are saved.
   - *fuzzbenchExpDir* points to the experiment_filestore set in the last section. Or set it to the experiment-data folder in the package to run this section with provided data.
   - *outDir* points to a path where final results are saved.
   - cores sets the number of concurrent tasks to run the triage.

   The dda.json file generated by the semantic matcher needs to be manually copied to outDir/[program name] for each program.
2. The triage scripts rely on python 3.9.0. The required packages can be installed with
   pip install -r RevBugBench/triage/requirements.txt

Run Triage

1. Run the triage process with
   python3.9 run_experiment.py -e exp -p [program name] -u --crash --coverage
   Explanations of each command line option can be found at
   https://github.com/SlaterLatiao/RevBugBench/blob/main/triage/README.md
   This will generate pickle files on [outDir]/[program name]/[fuzzer name], which can be later read and summarized in the next step. This process could take up to several days, according to the CPU frequency and number of cores used for the triage. Specifically, as mentioned in the 12th page of the paper, for libfuzzer-libxml2_reader and libfuzzer_libxml2_xml, a random size of 664 was taken in the evaluation, instead of triaging all the crashing inputs.
2. Finally, we can summarize the results by
   python3 run_experiment.py -p [program name] -e [experiment name] -c crashes coverage
   to generate a table like Table 2 in the paper.
   python3 run_experiment.py -p [program name] -e [experiment name] -g
   To generate an individual cause growth plot like Figure 9.
   python3 run_experiment.py -p [program name] -e [experiment name] -v
   To generate an individual cause venn diagram like Figure 10.
   Both figures will be saved on *outDir*.
3. We observed non-determinism on Step 4. This means running step 4 with provided Fuzzbench experimental results and then running step 5 may result in slightly different results than presented in the paper. Such differences do not affect the conclusions of the paper. To fully reproduce the results in the paper please check the next step.
4. The intermediate results of Step 4 generated in the paper evaluation are also provided. This data records for each fuzzing seed generated by the FuzzBench experiment, which

FixReverter injected bugs it reaches/triggers/crashes. The definition of reach/trigger/crash can be found at Section 3.4 and 3.5 in the paper. To use them, config *outDir* to be the *triage* folder in the package, and *fuzzbenchExpDir* to be the *experiment-data* folder. The following commands will reproduce Table 2, Figure 9 and Figure 10.

```
python3 run_experiment.py -e exp -c crashes coverage
python3 run_experiment.py -e exp -g
python3 run_experiment.py -e exp -v
```

Table 2 will be output in terminal, and figures can be found at the *outDir*.