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.
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.
6. **RevBugBench**. This folder stores a stable version of the RevBugBench repository (https://github.com/SlaterLatiao/RevBugBench)**.**

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 and kept on https://github.com/SlaterLatiao/RevBugBench. In this artifact we included a stable version of it (/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. 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 ..
   ```
4. Set up a config file following https://google.github.io/fuzzbench/running-a-local-experiment. An example can be found at https://github.com/SlaterLatiao/RevBugBench/blob/main/fuzzbench/experiment-config.yaml. 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 /RevBugBench/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, the reproduced results should be reasonably similar to the evaluation results (Table 2, Figure 9 and 10).
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 disk space.

RevBugBench 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 run_experiment.py -e exp -p [program name] -u --crash --coverage
   ```
   Explanations of each command line option can be found at /RevBugBench/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 slight non-determinism on the results of Step 1. It does not affect the conclusions of the paper, but will be investigated and fixed in the future development of RevBugBench.
4. The intermediate results of Step 1 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 /triage, and *fuzzbenchExpDir* to be /experiment-data. The following commands will produce the same numbers in 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.*

Running FixReverter on Other Programs

Following are the instructions to inject bugs on a program with FixReverter. If you want to try FixReverter on a program other than the 10 programs included in the artifact, please continue reading.

1. Create a FuzzBench benchmark with a C program following https://google.github.io/fuzzbench/developing-fuzzbench/adding-a-new-benchmark/, up to the *Testing it out* step.
2. Create a *clang* folder for the target program, like the programs included in the artifact.
    a. The *setup.sh* script is responsible for installing dependencies and downloading the target program. The *Dockerfile* created in step 1 can be used as a reference. It also copies the seeds folder onto /out/seeds.
    b. The *build.sh* script is responsible for building the target program. The *build.sh* created in step 1 can be used as a reference. As FixReverter requires a program's compilation database file, building with bear, compiledb or CMAKE may be necessary.
    c. The *build_cov.sh* script is based on build.sh with the following changes:
        i. The target program is built with extra flags. To be specific, the following flags need to be added to CFLAGS and CXXFLAGS -fsanitize=address,array-bounds,bool,builtin,enum,float-divide-by-zero,function,integer-divide-by-zero,null,object-size,return,returns-nonnull-attribute,shift,signed-integer-overflow,unreachable,vla-bound,vptr -DFRCOV
        ii. Fixes on the injected source code that fails the build. This is because FixReverter injects code based on a program's AST and cannot handle all the corner cases, like generated code.
    d. The *FixReverter.config* is the config file to run FixReverter Clang tools manager. The following fields need to be configured before running FixReverter:
        i. program source. It points to the directory of the target program.
        ii. entry file. It points to the file of the fuzzing target.
        iii. compilation database. It points to the compilation database file generated by *build.sh*.
        iv. seeds. It points to the seeds directory, usually /out/seeds.
        v. exe. It points to the executable of the fuzzing target.
        vi. build savers. It lists files that we want to inject the definitions instead of declarations of the FixReverter flag. It's used to fix some broken builds.
        vii. cores. It decides how many parallel tasks to run at the same time.
        viii. entry function. The entry-point of the fuzzing target. It's usually LLVMFuzzerTestOneInput.

    An example can be found at /artifact/FixReverter/drivers/inject/FixReverter.config. The other fields not mentioned above should not be changed.
3. Create a *phasar* folder for the target program, like the programs included in the artifact.
    a. The *build.sh* script installs the dependencies, downloads the program and builds it. At the end of *build.sh* /src/fuzzer.ll is generated by using extract-bc and llvm-dis commands on the built fuzzing target.

    b. When running the semantic matcher with image fr_phasar, the other 5 files, plugin.conf, psr_driver.c, psrFuzzingEngine.o, run_phasar.sh and .psrFuzzingEngine.o.bc are the same across programs, and can be found at /artifact/programs/lcms/phasar.
    c. When running the semantic matcher with image fr_phasar_port, the other 2 files plugin.conf and run_phasar.sh are the same across programs, and can be found at /artifact/programs/libxml2_xml/phasar.
    d. The image fr_phasar_port is only recommended to be used when fr_phasar runs into out-of-memory errors.
4. Modify the FuzzBench *Dockerfile* and *build.sh* to take the injected source code, for example, applying a patch of the injections. The file *benchmark.yaml* may also need to be updated, depending on whether the name of the project and fuzzing target have changed.
5. With the *clang*, *phasar* and FuzzBench benchmark folder, a fuzzer can be evaluated the same way as the provided 10 programs.