USENIX Security '24 Artifact Appendix: MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware

Michael Chesser
The University of Adelaide
Data61 CSIRO, Cyber Security
Cooperative Research Centre

Surya Nepal
Data61 CSIRO, Cyber Security
Cooperative Research Centre

Damith C. Ranasinghe
School of Computer Science
The University of Adelaide

A Artifact Appendix

A.1 Abstract

MULTIFUZZ is a new fuzzer for testing monolithic firmware targets that addresses the challenges of complex interactions between firmware and hardware in embedded devices. We evaluate MULTIFUZZ compared to the previous state-of-the-art monolithic firmware fuzzers Ember-IO and Fuzzware and also demonstrate MULTIFUZZ's ability to discover new bugs. We provide scripts for reproducing coverage benchmarks, including ablation studies, and for automatically running unit tests from synthetic example programs. Additionally, we include crashing inputs, replay scripts, and detailed analyses for all previously undiscovered bugs presented in the paper, as well as for false-positive crashes that were triaged during this process.

A.2 Description & Requirements

CPU Resources: In total running the full set of experiments requires approximately 580 CPU days of compute time, however this can be adjusted by modifying the benchmark configuration files to run fewer trials and/or fuzzer configurations.

RAM: Each trial typically requires less than 1 GB of RAM, however the exact amount may vary depending on the target and fuzzing configuration. Configurations without the 'Trim' stage enabled may use up to 8 GB of RAM.

Storage: Each trial will generate between 15 MB to 250 MB worth of temporary files. Then, the results from each trial are automatically compressed and copied to an output folder for post-processing and analysis. The final storage requirements for all experiments is approximately 1.5 to 2 GB (after temporary files are deleted).

To speed up benchmarking, our benchmark harness tool supports automatically executing multiple trials in parallel. The required RAM usage and storage for temporary files should be multiplied by the number of parallel workers selected for benchmarking.

A.2.1 Comparison to prior work

To compare against past work (Ember-IO, and Fuzzware), we followed the experimental procedures reported by the original authors from their GitHub repositories:

- https://github.com/fuzzware-fuzzer/fuzzware-experiments (02-comparison-with-state-of-the-art).
- https://github.com/Ember-IO/Ember-IO-Experiments (Fuzzing).

Generating the results needed for coverage comparison for each of the tools requires an additional 120 CPU days.

A.2.2 Security, privacy, and ethical concerns

During the setup process, network access is required to fetch from packages from github.com and crates.io and to download Docker images. Docker is used to isolate and run individual fuzzer runs, requiring the user to have Docker access.

We have already responsibly disclosed the previously unknown bugs listed in Table 3, however if any additional bugs are discovered during artifact evaluation, they should also be handled appropriately.

A.2.3 How to access

All the code and data required for running the experiments is open sourced on GitHub at:

- https://github.com/MultiFuzz/MultiFuzz-benchmarks/tree/usenix2024-ae

The benchmark repository uses Git Submodules to select a specific stable version of MULTIFUZZ for artifact evaluation.

A.2.4 Hardware dependencies

Our experiments require no specialized hardware, a generic x86-64 machine with enough RAM and storage is sufficient to run all benchmarks. In the paper, all experiments were conducted on a single AMD Ryzen Threadripper 3990X CPU with 256 GB of RAM running Ubuntu Server 20.04.

A.2.5 Software dependencies

Our experiments require a Linux-based OS with Bash, Docker, Rust and C compilers installed. Ghidra is optionally required for reproducing the metadata used for coverage analysis.

A.2.6 Benchmarks

We evaluate MULTIFUZZ on a collection of 20 firmware targets used in past work and additional 3 new firmware targets that we introduced in our evaluation. We include pre-built binaries for all of the firmware targets used in benchmarks. The new firmware targets can optionally be bit-for-bit reproduced by running the `./build_new_binaries.sh` from the `benchmarks` sub-directory.

To fairly compare the coverage, we normalize the coverage results from each fuzzer by filtering blocks according to control-flow metadata obtained using a Ghidra script. The control-flow metadata files can be reproduced by running the Ghidra script (ExportCfg.java) after loading each binary into Ghidra.

A.3 Set-up

The installation procedure has been tested on a plain Ubuntu Server 22.04, however any modern x86-64 Linux distribution with support for Rust and Docker is likely to work.

A.3.1 Installation

First, ensure that the current user has access to Docker and following dependencies are installed: Rust (tested with rustc 1.75.0), Clang, Docker, libssl-dev, pkg-config, and libfontconfig-dev (plotting only).

On a Ubuntu 22.04 server this can be done by running the following commands:

```
curl https://sh.rustup.rs -sSf | bash -s -- -y
. "$HOME/.cargo/env"
sudo apt update
sudo apt install -y clang docker pkg-config \
    libssl-dev libfontconfig-dev
sudo usermod -aG docker $USER
newgrp docker
```

After these dependencies have been installed, run the `./build_all.sh` script to compile and build MULTIFUZZ, the benchmark harness tool, and the post-processing and analysis tool.

Optionally, if you wish to follow along with the crash analysis steps in the bug analysis experiment, you may want to download a copy of the GNU Arm Embedded Toolchain so that `arm-none-eabi-gdb` is available.

A.3.2 Basic Test

The simplest way to test that the fuzzer has been compiled and built correctly, test that the fuzzer is able to successfully replay a known input. This can be done by running the following command:

```
./replay.sh crashes/Gateway/zero_length_sysex
```

The output of the script should include a line containing the message: '`UnhandledException(code=ReadUnmapped, value=0x800080)`', along with additional information for debugging.

To verify that the benchmarking tool and every fuzzer configuration is fully functional, a fast benchmark profile (approximately 20 CPU minutes) is available by running:

```
./benchmark-test.sh [no. parallel tasks to use]
```

After execution, the `./bench-harness/output/multifuzz/` directory should contain several sub-directories: `debug-all`, `debug-extend`, `debug-havoc`, `debug-i2s`, `debug-trim`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): MULTIFUZZ is able to fuzz a wide-range of firmware targets and hardware, either on par or better than existing state-of-the-art fuzzers Ember-IO and Fuzzware. This corresponds to evaluation in Section 5.1 and Section 5.2 of the paper, and is proven in the experiments (E1) and (E2). The results of these experiments are used to generate Figure 11 and Table 2 in the paper.

(C2): Each of the stream-orientated input generation techniques introduced by MULTIFUZZ have an impact on the fuzzer's effectiveness. This corresponds to Section 5.3 and Appendix A.2 of the paper, and is proven in experiment (E3). The results of these experiments are used to generate Table 1 and Figures 12, 13 and 14 in the paper.

(C3): MULTIFUZZ has been used to uncover 18 new bugs across 8 binaries (Table 3 in the paper). This is demonstrated in the Section 5.4

A.4.2 Experiments

(E1): P^{2}IM unit-tests [5 human-minutes + 8 CPU Hours]: This experiment runs MULTIFUZZ on the 46 unit test binaries from P^{2}IM for a short period of time (10 mins

per binary) and verifies that the fuzzer is able to find inputs that satisfy each unit test.

Preparation: Ensure that the setup script from A.3.1 has been executed. Navigate to the `p2im-unittests` directory.

Execution: From within the `p2im-unittests` directory run the `./run.sh` script. The script accepts an optional argument that controls how many tests to run in parallel across multiple cores.

Results: On successful completion, the string: "`66 successes, 0 errors`" will be printed to stdout and additional logging information is printed to stderr describing which inputs satisfy each unit test. Additionally, the `workdir-p2im-unittests` sub-directory contains fuzzer metadata for each trial.

(E2): Code-coverage Evaluation [20 human-minutes + 130 CPU days]: This experiment uses MULTIFUZZ to fuzz 23 real-world ARM firmware for 24 hours (repeated for 5 times) and evaluates the code-coverage reached over-time.

Preparation: Ensure that the setup script from A.3.1 has been executed.

Execution: Run the `./multifuzz-coverage.sh` script. The script accepts an optional argument that controls how many tests to run in parallel across multiple cores.

Results: On successful completion of the benchmark the message "`All tasks complete`" is printed to stderr, and the results of the benchmark are saved to `./bench-harness/output/multifuzz-all`. The results can then be post-processed and analyzed by running `./analyze-results.sh`. A coverage graph similar to Figure 11 from the paper is saved to `./analysis/output/coverage.svg`, and Table 2 can be reproduced using the data from `./analysis/output/total_blocks_per_trial.csv`.

(E3): Ablation Study [20 human-minutes + 470 CPU days]: This experiment tests the fuzzing ability of MULTIFUZZ with various features disabled[1] on the same set of binaries as E2.

Preparation: Ensure that the setup script from A.3.1 has been executed.

Execution: Run the `./multifuzz-ablation.sh` script. The script accepts an optional argument that controls how many tests to run in parallel across multiple cores.

Results: On successful completion of the benchmark the message "`All tasks complete`" is printed to stderr, and several new sub-directories containing the results are located at `./bench-harness/output/multifuzz-*`. Table 1 of the paper can be reproduced from `./analysis/output/median_coverage.csv` which is generated by running `./analyze-results.sh`.

(E4): Previously Undiscovered Bugs [1 to 2 human-hours]: This experiment validates that MULTIFUZZ finds new previously undiscovered bugs, and that the bugs discovered are real bugs and not false-positives.

Preparation: Ensure that the setup script from A.3.1 has been executed.

Execution: Executing this experiment involves using the `./replay.sh` script to replay crashing inputs corresponding to each bug in the paper, and following the analysis described in the analysis.md document to verify each crash. Additionally, MULTIFUZZ supports attaching a debugger (GDB) to perform more fine-grained analysis on the exact reason for each crash.

Results: Replaying each input should result in the fuzzer reporting a crash (e.g. `UnhandledException(code=ExecViolation, value=0x3f3e3d3c)`) along with additional metadata for debugging the crash including an *approximate* stack trace (GDB is required be used to obtain a precise stack trace) and a list of the last 10 blocks executed. The output should match the analysis described in the `analysis.md` document.

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/usenixsec2024/`.

[1] Note: The configurations where the 'trim' stage is disabled may use a large amount of RAM (e.g., 8 GB), so care should be taken when running multiple trials in parallel.