

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FirmReBugger: A Benchmark Framework for Monolithic Firmware Fuzzers

Mathew Duong, Michael Chesser, and Guy Farrelly, *University of Adelaide;* Surya Nepal, *Data61 CSIRO;* Damith C. Ranasinghe, *University of Adelaide*

https://www.usenix.org/conference/usenixsecurity26/presentation/duong

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology



USENIX Security '26 Artifact Appendix: FIRMREBUGGER: A Benchmark Framework for Monolithic Firmware Fuzzers.

Mathew Duong[1], Michael Chesser[1], Guy Farrelly[1], Surya Nepal[2], and Damith C. Ranasinghe[1]

[1]*University of Adelaide*
[2]*Data61 CSIRO*

A Artifact Appendix

A.1 Abstract

FIRMREBUGGER is a bug-oracle–based evaluation framework and benchmark suite for monolithic firmware fuzzers that enables automated, fair, and reproducible assessment of bug discovery. It supports hard-to-measure metrics such as time-to-bug, which are computed, visualized, and reported using C-based bug oracles included with the artifact. The fuzzers evaluated in the paper are provided with Dockerfiles and automated build scripts to facilitate reproduction. Additionally, the artifact includes a web-based interface for scheduling and managing fuzzing runs and for visualizing the reported metrics.

A.2 Description & Requirements

The full evaluation of all nine state-of-the-art monolithic firmware fuzzers reported in the paper requires approximately 10 CPU-years. Due to this substantial computational cost, the number of trials, fuzzing duration, and selection of fuzzers and binaries can be adjusted as needed during artifact evaluation. Hardware requirements vary by fuzzer, but in general each fuzzing instance requires approximately 2 GB of RAM. For CPU resources, any modern x86 processor is sufficient, with one fuzzer instance typically assigned per physical CPU core.

A.2.1 Security, privacy, and ethical concerns

During setup, network access is required to fetch dependencies from external sources (e.g., GitHub). Docker is used to isolate and execute individual fuzzer runs and therefore requires a functional Docker environment. All previously unknown bugs have been responsibly disclosed. Any additional bugs discovered during artifact evaluation should be handled following responsible disclosure guidelines.

A.2.2 How to access

All code and data are publicly available at:

- **FirmReBugger stable release:** https://figshare.com/s/d37c9a1cb67b6d07be0c

We also have a GitHub repository, with the version corresponding to the USENIX submission provided as a tagged release:

- **FirmReBugger GitHub repository:** https://github.com/FirmReBugger/FirmReBugger/releases/tag/USENIX-Artifact-Evaluation

A.2.3 Hardware dependencies

The experiments do not require specialized hardware and can be run on a generic x86-64 system with adequate RAM and storage. The results reported in the paper were obtained on an AMD Ryzen Threadripper 5995WX system with 256 GB of RAM running Ubuntu Server 22.04.

A.2.4 Software dependencies

Our experiments require a Linux-based operating system with the following software dependencies: Bash, Docker, C compiler, Python, npm and UV.

The evaluation also depends on the following fuzzing frameworks and tools:

- Ember-IO-Fuzzing
- Fuzzware
- Fuzzware-Icicle
- SEmu
- DICE
- Hoedur
- GDMA
- SplITS
- Hoedur

A.2.5 Benchmarks

The artifact includes the FIRMBENCH benchmark, consisting of 61 real-world monolithic firmware binaries and 280 documented bugs, as well as C-based bug oracles. This data is required to reproduce the experiments reported in the paper.

A.3 Set-up

The setup has been tested on a clean installation of `Ubuntu Server 22.04`. Certain fuzzers require adjustments to system parameters. Specifically, enable core dumps and increase the limits for inotify watches and instances, which are necessary for the operation of some fuzzers included in this artifact. Note that `Docker` may require `sudo` privileges, unless your user has been added to the `docker` group. For more details, refer to the README at https://github.com/FirmReBugger/FirmReBugger.

A.3.1 Installation

After ensuring the initial setup, the following software must be installed:

- `uv` (https://docs.astral.sh/uv/getting-started/installation/)
- `docker` (https://docs.docker.com/engine/install/)
- `npm` (version $\geq$ 22.0.0; see https://docs.npmjs.com/downloading-and-installing-node-js-and-npm)

In addition, please install the following packages:

```
Install uv https://docs.astral.sh/uv/
    getting-started/installation/
Install docker https://docs.docker.com/
    engine/install
sudo apt-get install automake texinfo
    unzip
sudo apt install gcc-arm-none-eabi
sudo apt install texlive-latex-base
sudo apt install texlive-xetex texlive-
    fonts-recommended texlive-latex-
    extra
install npm (npm >= 22.0.0)
```

And set up the host for fuzzing: Set up the host for fuzzing:

```
echo core | sudo tee /proc/sys/kernel/
    core_pattern
sudo sh -c echo 524288 > /proc/sys/fs/
    inotify/max_user_watches && echo
    8192 > /proc/sys/fs/inotify/
    max_user_instances
```

A.3.2 Build All Fuzzers

If you're having trouble building the fuzzers locally or do not need to, you can pull our prebuilt images from GHCR:

```
uv run frb build --use-prebuilt
```

This pulls registry images and tags them locally as `frb:<fuzzer>` and `frb_original:<fuzzer>`, which are the names used by fuzzing/triage runtime.

Fall back to build locally If the prebuilt images fail to run on your system, fall back to the non-prebuilt images with:

```
cd FirmReBugger
export FIRMREBUGGER_BASE_DIR=$(pwd)
uv run frb build
```

A.3.3 Web Application (Recommended Approach)

Launch the web application with the following commands:

```
uv run frb app --help
uv run frb app -p <port>
```

- **Report:** Provides a summary of your fuzzing campaigns using FirmReBugger.
- **Job Manager:** Allows you to configure and schedule fuzzing or triaging jobs automatically.

A.3.4 Available Commands

The following commands are available for reference:

```
cd FirmReBugger
export FIRMREBUGGER_BASE_DIR=$(pwd)
uv run frb --help
uv run frb fuzz --help
uv run frb build --help
uv run frb bug-analyzer --help
uv run frb charting-tool --help
uv run frb app --help
```

A.3.5 Basic Test

To verify your installation, first build a few fuzzers using the prebuilt images:

```
uv run frb build --use-prebuilt
```

Next, start the web application (served on `localhost:5000`; if you are connected remotely, you may need to forward the port):

```
uv run frb app
```

The web interface provides a summary report of your active and completed fuzzing campaigns using FirmReBugger, including interactive visualizations and management tools.

The *Job Manager* allows you to configure and schedule fuzzing or triaging jobs.

Recommended usage: Launch fuzzing and triaging directly through the web app interface. This ensures that inputs, job scheduling, and result visualization are handled automatically.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *A comprehensive evaluation of 7 state-of-the-art monolithic firmware fuzzers (Ember-IO, Fuzzware, Fuzzware-Icicle, SEmu-Fuzz, SplITS, Hoedur, and MultiFuzz) was conducted on the FirmBench dataset comprising 31 binaries. The results of this evaluation can be found at our GitHub repository[1], and underlying data is used in Figures 9, and 12 of the paper.*

(C2): *We benchmarked 6 state-of-the-art monolithic firmware fuzzers (Ember-IO, Fuzzware, Fuzzware-Icicle, SplITS, Hoedur, and MultiFuzz) on the FirmBenchX dataset, comprising 22 additional binaries. These results can be accessed at our GitHub repository[2], and the corresponding analysis is presented in Figures 10, and 12.*

(C3): *We evaluated two state-of-the-art DMA-centric firmware fuzzers (GDMA and DICE) on the FirmBenchDMA dataset, which includes two binaries. These results are summarized in Table 1 and contribute to the analyses found in Figure 10.*

A.4.2 Experiments

(E1): **How to:** Open the FirmReBugger web app and use the Job Manager to schedule fuzzing jobs for the FirmBench suite using each of the 7 fuzzers. Configure each job for 10 trials per binary, each running for 24 hours. Once fuzzing is complete, queue triaging jobs for each fuzzer via the "More Actions" button in the completed section. Visualize the results using the Report page.
Preparation: Ensure that all dependencies and environment configurations described in Section A.3.1 have been completed.
Execution: In the FirmReBugger base directory, launch the web app with:

```
uv run frb app
```

Open the web browser at `localhost:5000`, navigate to "Job Manager," and schedule fuzzing jobs for all FirmBench binaries, assigning each of the listed fuzzers.
Results: After both fuzzing and triaging are completed, review results on the Results/Report page. The outputs should qualitatively align with those reported in the GitHub repository and match summary data in Figures 9, 10, and 12 of the paper.

(E2): **How to:** Use the Job Manager in the FirmReBugger web app to schedule fuzzing and triaging for the FirmBenchX benchmark (22 binaries) using Ember-IO, Fuzzware, Fuzzware-Icicle, SplITS, Hoedur, and MultiFuzz, with 10 trials per binary for 24 hours each. Visualize all results as in E1.
Preparation: As above, confirm all dependencies and environmental setup from Section A.3.1.
Execution: Launch FirmReBugger, open the web interface, and repeat the scheduling and triage process for the FirmBenchX suite and listed fuzzers.
Results: Review results in the Report page, and compare with published results on GitHub and in Figures 9, 10, and 12.

(E3): **How to:** In FirmReBugger's Job Manager, schedule fuzzing and triage tasks for GDMA and DICE against the two FirmBenchDMA binaries, with recommended durations and trial counts. Inspect and visualize the results as with the other benchmarks.
Preparation: Complete all installation and environment setup as per Section A.3.1.
Execution: Launch the web app, open the Job Manager, and schedule the relevant jobs for the DMA-centric evaluation.
Results: Result summaries will be available through the Reports page; validate your findings against Table 1 in the paper.

A.5 Notes on Reusability

FirmBench is designed to be a living and extensible benchmarking platform for firmware fuzzing. We encourage and support continued community contributions, enabling the benchmark suite to grow and remain relevant as new fuzzing techniques and bug types emerge.

The artifact provides mechanisms for users to submit new bugs ("Ravens") and new binaries, together with a thorough root cause analysis, for potential inclusion in FirmBench. Community review and validation ensure that each Raven accurately encapsulates the bug semantics before being merged. This process maintains the integrity and utility of the benchmark for future research.

As the field evolves, FirmBench allows for the inclusion of focused subsets, such as new categories of bugs or benchmarks guarded by specific "roadblocks" (e.g., DMA-centric bugs), to support evaluation of targeted fuzzing techniques. The infrastructure is flexible and can accommodate the creation of balanced binary sets with diverse bug types and challenge properties.

A.6 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/usenixsec2026/.

[1] https://github.com/FirmReBugger/FirmReBugger/blob/main/paper_results/FirmBench.pdf

[2] https://github.com/FirmReBugger/FirmReBugger/blob/main/paper_results/FirmBenchX.pdf