Forming Faster Firmware Fuzzers

Lukas Seidel, *Qwiet AI;* Dominik Maier, *TU Berlin;*
Marius Muench, *VU Amsterdam and University of Birmingham*

https://www.usenix.org/conference/usenixsecurity23/presentation/seidel

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

USENIX'23 Artifact Appendix: Forming Faster Firmware Fuzzers

Lukas Seidel[1], Dominik Maier[2], and Marius Muench[3]

[1]*Qwiet AI, jlseidel@qwiet.ai*
[2]*TU Berlin, dmaier@sect.tu-berlin.de*
[3]*VU Amsterdam and University of Birmingham, m.muench@bham.ac.uk*

A Artifact Appendix

A.1 Abstract

This artifact allows the replication of the experiments and results described in Section 6. We provide the following: (i) A stand-alone repository containing the full source code for our rehosting and fuzzing engine, ready to be compiled and used (https://github.com/pr0me/SAFIREFUZZ), (ii) a repository containing documentation, build- and setup scripts for replicating our experiments and a copy of the data we gathered during our evaluation (https://github.com/pr0me/safirefuzz-experiments).

The artifact has been validated on a HoneyComb LX2 ARM workstation containing 16 ARM Cortex-A72 cores with a clock rate of up to 2 GHz, 32 GB DDR4 memory with a frequency of 3200 MT/s and a 128 GB m.2 SSD running Ubuntu 18.04.05.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

While running and evaluating SAFIREFUZZ does not require destructive steps, small changes to the host system weakening its security guarantees are needed to run our system.

In particular, we require ASLR to be disabled, to increase determinism and avoid mapping of, e.g., linked libraries in segments we need otherwise and expect to be empty. Additionally, we enable allocating virtual memory down to address 0 by adjusting `mmap_min_addr`, as we need to place parts of the firmware image in low memory regions. Both can be configured by running the `SAFIREFUZZ/prepare_sys.sh` script. Those changes should be reverted after usage of our system, either by manually reverting the changes or rebooting.

A.2.2 How to Access

We provide public access to our code and experiment setups and data through the following GitHub repositories at specific tags for artifact evaluation:

1. SAFIREFUZZ main repository: https://github.com/pr0me/SAFIREFUZZ/tree/post_ae
   DOI: https://zenodo.org/record/8223057
2. Artifact Evaluation data: https://github.com/pr0me/safirefuzz-experiments/tree/post_ae
   DOI: https://zenodo.org/record/8223055

The repositories contain detailed information on building, running, and reproducing our experiments.

A.2.3 Hardware Dependencies

SAFIREFUZZ rehosts low-level Cortex-M firmware onto more powerful Cortex-A cores. As such, a system containing a Cortex-A core with 32-bit support is required, which can for instance be found on a Raspberry Pi 4b featuring four Cortex-A72 cores. Installation instructions for Raspberry Pis can be found in our main repository. Additionally, due to interoperability issues, our Fuzzware-specific experiments were run on an x86-64 VM.

During artifact evaluation, we provided the reviewers with access to the same hardware we used during our evaluation: (**M1**) a *HoneyComb* ARM workstation, and **(M2)** an Ubuntu 18.04 x86-64 VM hosted on an AMD EPYC 7662 server.

A.2.4 Software Dependencies

1. **Rust:** Our artifact is implemented in the Rust programming language. Per the `rust-toolchain` file provided in the main repository [1], we pin the installation environment to compiler version `rustc 1.62.0-nightly`.
2. **Cross-Compilation:** A cross compilation toolchain is required. On Ubuntu, the corresponding packets are `gcc-arm-linux-gnueabihf` and `g++-arm-linux-gnueabihf`.

   Install the `armv7-unknown-linux-gnueabihf` rust target for the above-mentioned compiler version. Note that these steps are even required when directly building in an ARM environment such as the HoneyComb. While the processor can execute programs targeted for both

ARMv7 and ARMv8 versions, if the OS is built for aarch64, cross-compilation is required as the artifact binary will execute in ARMv7's 32-bit mode.

3. **External Dependencies:** The main artifact requires the *LibAFL* and *Keystone* external dependencies that cannot be automatically fetched by Rust's package manager. We include the dependencies as git submodules, pinned to specific versions.
   The evaluated third-party frameworks introduce their own dependencies and can be set up as documented in the HALucinator[1] and Fuzzware[2] repositories.

4. **Python:** For multiple build and automation scripts provided with the AE experiment repository, we require a Python version > 3.9. Additionally, we require the following Python libraries for analyzing and plotting the results of our experiments: `jupyter`, `numpy`, `matplotlib`, `seaborn`, `scipy`, `pandas`.

A.2.5 Benchmarks

To evaluate SAFIREFUZZ, we use a collection of 12+2 firmware samples: 12 samples from the original HALucinator evaluation, and 2 previously untested samples (*JPEG Decoder* and *STM32 Sine*). We include all samples in the experiment repository under `00_firmware`.

Using these samples, we evaluate our approach against the following fuzzing setups:

1. **HALucinator.** State-of-the-art high-level-emulation-based rehosting and fuzzing framework. We include the fuzzing-ready *hal-fuzz* version as a submodule in `safirefuzz-experiments/01_fuzzing/hal-fuzz`.

2. **HALucinator - LibAFL.** We replace HALucinator's legacy AFL forkserver with a LibAFL-based forkserver. This new version is identical in configuration to the forkserver backend we use in SAFIREFUZZ. We conduct this comparison to eliminate variables such as differences in mutation strategies. Details can be found in the *safirefuzz-experiments* repository under `01_fuzzing/forkserver_LibAFL`.

3. **Fuzzware.** A recent peripheral-modeling-based rehosting approach. This is the only experiment we conducted in an x86-64 environment, as, even after consulting the authors, Fuzzware could not be brought to run in our default ARM environment. We provide usage information and link the necessary submodule under `01_fuzzing/fuzzware`.

We include setup guides and detailed usage instructions for all evaluated frameworks under `01_fuzzing/README.md`.

[1] https://github.com/ucsb-seclab/hal-fuzz
[2] https://github.com/fuzzware-fuzzer/fuzzware

A.3 Set-up

A.3.1 Installation

If you are using the provided access to the experiment machines, all systems are already set-up and below instructions can be skipped. To manually install SAFIREFUZZ, the main artifact, please follow these steps:

1. Checkout our experiments repository 2 and initialize the submodules recursively.
2. Inside the experiments repository:
`$> cd 01_fuzzing/SAFIREFUZZ`
3. Install the Rust programming language.[3]
4. Install the cross-compilation toolchain with `'rustup target add armv7-unknown-linux-gnueabihf'` and cross-arch linkers, e.g., on Ubuntu by running `'sudo apt install gcc-arm-linux-gnueabihf g++-arm-linux-gnueabihf'`.
5. Specify the correct linker by adding the following lines to your `~/.cargo/config`:
```
[target.armv7-unknown-linux-gnueabihf]
linker = "arm-linux-gnueabihf-gcc"
```
6. Specify the target harness you want to execute / fuzz in `src/engine.rs`:
`use crate::harness::wycinwyc as harness;`
7. Build with
`$> cargo build -release -target armv7-unknown-linux-gnueabihf`
8. Run the `prepare_sys.sh` script as root.

A.3.2 Basic Test

After installing and compiling the main artifact, you will find the `safirefuzz` binary under `./target/armv7-unknown-linux-gnueabihf/release/`. Compilation is always specific to a single target or harness, so make sure to change the target (cf. Section A.3.1, step 6.) and re-compile before trying to execute a new firmware image.

Start fuzzing a specific firmware image with a directory of seeds by running:

```
./safirefuzz -b 00_firmware/wycinwyc.bin -i
01_fuzzing/seeds/wycinwyc/ -c 1.
```

When starting a fuzzing campaign, you should see LibAFL's status reports scrolling by. For running a test on the WYCINWYC target, you should be able to see rapidly increasing numbers for *corpus*, around 400-600 after approx. 30 seconds, which are interesting inputs leading to unique new coverage, at roughly 7000 executions per second. You can find these inputs in the *queue* directory while crashing inputs are stored in *crashes*.

To then execute a single input, execute:

[3] https://www.rust-lang.org/tools/install

```
./safirefuzz -b 00_firmware/wycinwyc.bin -i
01_fuzzing/crashes/SOMECRASHID
```

We automated most of these steps with the `safirefuzz_target.py` script included in the experiments repository under `01_fuzzing`. For instance, running `$> ./safirefuzz_target.py nxp_http` will automatically build SAFIREFUZZ for the correct target and start fuzzing. This script defaults to running on the third core (`-c 2`), change this if you are running multiple tests in parallel.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): SAFIREFUZZ achieves statistically significant more exec/s (ca. 690x on avg.) and coverage than HALucinator, except for coverage on the P2IM PLC and P2IM Drone targets. This is proven by experiment E1.

(C2): SAFIREFUZZ achieves more exec/s (ca. 1100x on avg.) and coverage than HALucinator-LibAFL, except for coverage on the UDP Echo Server, STM PLC, P2IM PLC and P2IM Drone targets. These results are statistically significant, except for coverage on the P2IM PLC, STM PLC, WYCINWYC, and UDP Echo Client targets. This is proven by experiment E2.

(C3): SAFIREFUZZ achieves more exec/s (ca. 145x on avg.) and coverage than Fuzzware, except for coverage on P2IM PLC[4] and P2IM Drone. The results are statistically significant except for coverage on the 6LoWPAN RX/TX, STM PLC, and WYCINWYC targets and for execution speed on the SAMR21 target. This is proven by experiment E3.

(C4): SAFIREFUZZ reliably re-discovers previously found bugs during fuzzing (E0). This includes vulnerabilities in the WYCINWYC and 6LoWPAN RX/TX targets as discussed in Section 6.4.

(C5): SAFIREFUZZ finds crashes in the previously untested firmware images *JPEG Decoder* and *STM32 Sine*. This can be replicated with experiment E4. We discuss the findings in Section 6.4 of our paper.

For C1, C2 and C3, we discuss our results in Section 6.2 in the main paper. Table 3 reports numbers gathered during our experiments and Figure 3 illustrates achieved coverage over the course of a 24-hour fuzzing campaign for all targets and frameworks.

[4]The paper as published as part of the proceedings contains an error in which the list of valid basic blocks for the P2IM PLC target was calculated incorrectly. This led to underreporting achieved coverage, impacting Fuzzware the most. We would like to thank Chris Boyce for pointing this out, based on the experiment data we published. A version with updated numbers and graphs can be found under https://download.vusec.net/papers/safirefuzz_sec23.pdf.

A.4.2 Experiments

As a working SAFIREFUZZ installation is required for the subsequent steps, refer to Section A.3.1 of this Appendix and the README of our main repository [1] for instructions.
The following steps assume you work in the pre-configured environments.

(E0): SAFIREFUZZ Baseline [20 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 10 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: Use the `./safirefuzz_target.py` script in `01_fuzzing` of the experiments repository [2] to start a 24-hour fuzzing campaign for the specified target with SAFIREFUZZ.

(E1): HALucinator Comparison [20 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 10 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: HALucinator is readily set-up, you can start fuzzing with this framework by executing the corresponding script in the *hal-fuzz* submodule. For further details, refer to the HALucinator section in `01_fuzzing/README.md`.

(E2): HALucinator-LibAFL Comparison [20 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 10 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: For details how to start a HALucinator-LibAFL fuzzing campaign, refer to the corresponding section in `01_fuzzing/README.md`.

(E3): Fuzzware Comparison [15 human-minutes fuzzing set-up time + up to 5x12x24 compute-hours + 15 human-minutes coverage collection set-up time + 2-6 compute-hours queue replay time]: In order to set up and start fuzzing with Fuzzware, please refer to the detailed instructions provided `01_fuzzing/fuzzware` as part of our experiments repository.

(E4): Vulnerability discovery [15 human-minutes fuzzing set-up time + up to 24 compute-hours + 15 human-minutes replay & verification]: To compile and fuzz the previously untested targets, please refer to the `README` included in `03_case_studies` inside the experiment repository.

Collecting Coverage. For all experiments except E3, coverage can be collected using the `eval_bbs_halucinator.py`

script in `02_coverage_collection`. For detailed instructions on this, refer to the `README` provided within the directory. For E3, use the scripts `fuzzware_genstats_with_hal.sh` and `fuzzware_genstats_without_hal.sh` provided in `02_coverage_collection` and refer to the `README` for details.

Analyzing Results. We provide scripts to test whether achieved coverage and execution speeds are statistical significant under `04_eval_data` inside the experiment repository. Please use the `bb_mann_whitney.ipynb` and `execs_mann_whitney.ipynb` jupyter notebooks inside the `coverage` and `executions` directories. We further provide a `gen_fig3.ipynb` notebook to plot coverage data over time. To use these notebooks with data from your experiments, you will need to exchange the `.data` and `.csv` in the according according subdirectories. Please refer to the `README` for more details.

Time & Resource Considerations. Due the extent of the experiments carried out during evaluation, it may not be possible to run all experiments for all reviewers in the time frame allocated for artifact evaluation. Hence, we provide the raw data collected from our runs under `04_eval_data` in our experiment repository. The raw data allows to reproduce our claims without, or only partially, running the experiments.

A.5 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/.