Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing

Tobias Scharnowski, Nils Bars, and Moritz Schloegel, *Ruhr-Universität Bochum;*
Eric Gustafson, *UC Santa Barbara;* Marius Muench, *Vrije Universiteit Amsterdam;*
Giovanni Vigna, *UC Santa Barbara and VMware;* Christopher Kruegel,
UC Santa Barbara; Thorsten Holz and Ali Abbasi, *Ruhr-Universität Bochum*

https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

A Artifact Appendix

A.1 Abstract

FUZZWARE is a firmware emulation and fuzzing prototype which makes use of symbolic execution to model MMIO accesses. In our experiments, we fuzz tested different sets of samples (synthetic, state-of-the-art, and new targets for CVE discovery). Based on the experiment stage (analogous to our paper's evaluation subsections), we collect additional data such as modeling statistics, job timings, unit test coverage, and code coverage.

As a fuzzing work, our experiments require computational resources. At a minimum (for a single-iteration evaluation of our core experiments instead of the 5/10 iterations that we performed), you should expect to perform 42 CPU days worth of fuzzing time on a single Linux system during the evaluation period (21 for the state-of-the-art comparison only). For the easiest (and repeated) reproduction, we recommend 41 dual-core Ubuntu cloud VMs (2 CPUs / 4-6GB RAM / 64 GB storage) which will run for 11 days to perform the full replication. Other setups are possible, but will require a bit of tinkering with experiment run scripts.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Locally-scoped Dynamic Symbolic Execution
- **Program:** Fuzzware builds on top of AFL/AFL++, unicorn engine, angr (8.19.10.30), Python < 3.10 (due to angr version)
- **Compilation:** clang
- **Binary:** Firmware samples used for evaluation
- **Data Set:** Included: P2IM Unit Tests, P2IM Targets, uEmu Targets, Artificial Password Firmware Samples, Contiki-NG & Zephyr-OS Target Samples
- **Run-time environment:** Linux, Docker
- **Hardware:** The recommended setup for full replication requires 41 dual-core Ubuntu cloud VMs.
- **Metrics:** Unit Test Coverage, Model Generation Timings, MMIO Overhead Elimination Statistics, Code Coverage, Reached Bugs
- **Output:** Included: Crashes (binary files), Generated: Fuzzing Inputs (binary files), MMIO Models (text files), statistics (text files), GNU plots (PNG files)
- **Experiments:** bash scripts, readmes
- **How much disk space required (approximately)?:** Recommended setup: 25 GB of local storage for collected experiment results, and 41 Ubuntu cloud machines with 64GB storage each. For a fully local setup (run script customizations are required), 100GB should suffice to hold all experiment data.
- **How much time is needed to prepare workflow (approximately)?:** 4h
- **How much time is needed to complete experiments (approximately)?:** 5-10 days (on 41 Ubuntu cloud machines, total CPU time: 320 days for full experiment repetition count, 42 days for a single iteration)
- **Publicly available (explicitly provide evolving version reference)?:** Evolving: https://github.com/fuzzware-fuzzer
- **Publicly available?** Yes. Stable version: sec22-ae-accepted
- **Code licenses (if publicly available)?:** Apache-2.0
- **Data licenses (if publicly available)?:** Apache-2.0
- **Archived (explicitly provide DOI or stable reference)?:** 10.5281/zenodo.6499215

A.3 Description

A.3.1 How to access

We release both the research prototype, as well as all experiments and data as open source on GitHub at the following two locations:

- https://github.com/fuzzware-fuzzer/fuzzware/tree/sec22-ae-accepted
- https://github.com/fuzzware-fuzzer/fuzzware-experiments/tree/sec22-ae-accepted

A.3.2 Hardware dependencies

For the experiment reproduction, x86 computation resources are required. For the easiest reproduction (no customization of run scripts), 41 dual-core Ubuntu cloud instances are recommended (2 cores, 4-6GB RAM, 64GB storage). With run script customizations, other hardware setups that allow for 320 days worth of fuzzing computation time within a reasonable time frame can be used. In case a single-run reproduction is deemed sufficient, a total of 42 days worth of computation time (plus some compute for trace generation and metric aggregation) are required.

A.3.3 Software dependencies

We recommend Linux/Docker as the experiment platform for a reproducible setup of all dependencies. For a seemless reproduction, we further recommend an Ubuntu LTS release (e.g., 18.04 or 20.04). Note that the version of angr which is pinned for the evaluation constrains the python version to be <3.10, which means that Ubuntu LTS releases of coming years may require installing an older version of python than are the default for future Ubuntu releases.

A.3.4 Data sets

We include all required firmware samples for running the experiments in the GitHub repository. In more detail, we include a list of target firmware images from previous work (P2IM, uEmu), and compiled additional targets for bug discovery, which are present as pre-built binaries in the fuzzware-experiments repository.

To reproduce our newly introduced target firmware samples, we further provide build scripts for all relevant targets.

A.3.5 Models

We do not include the MMIO models generated by Fuzzware. These will be generated by the prototype on-the-fly during the experiments.

A.3.6 Security, privacy, and ethical concerns

The fuzzware-experiments repository contains crash cases for security critical bugs in ZephyrOS and Contiki-NG. All corresponding vulnerabilities have been disclosed to the maintainers and patches were developed.

A.4 Installation

Installing individual instances of Fuzzware is done via the build_docker.sh script (for a Docker-based setup), and via the install_local.sh script (for a native setup), which are both located in the fuzzware repository.

To ease the setup process for the experiments, we also created scripts in the fuzzware-experiments repository. to remotely install SSH-accessible Ubuntu instances. The corresponding script can be found in ssh_hosts_install.py. The README files within the repository and comments in the source code are meant to provide additional information to allow for the use of scripts and the customization of the installation process.

The recommended setup is to create 41 Ubuntu LTS hosts with 2 cores, 6GB RAM, and 64 GB of disk space each. This could be reduced 4 GB RAM and 32 GB disk space in case costs require minimizing.

A.5 Experiment workflow

The data required to conduct each experiment is contained within the fuzzware-experiments repository. The repository is organized in a way such that each subdirectory corresponds to a particular section in the paper. The mapping from directory to paper can be found in the top-level README.md file.

In essence, each experiment entails a 24-hour fuzzing run of the target (invoked via the "fuzzware pipeline" utility), which creates a *fuzzware-project* directory. This directory contains the state of the MMIO model configurations, as well as inputs that were generated by fuzzers over the span of the fuzzing run. In a second step, metrics such as code coverage are aggregated from this raw data. Finally, in a third step, depending on the experiment, additional aggregation is performed over the full set of fuzzing runs belonging to the given experiment. This aggregation collects a summary of the data as can be found in the respective section of the paper. Below we describe the workflow for each of these experiments.

(1) PW discovery & Unit Tests. For the first experiment, the following workflow can be used to reproduce the experiments:

1. Make sure to have installed fuzzware on cloud hosts via ssh_hosts_install.py. If 41 dual-core hosts have been installed with the expected naming conventions, no modifications to run scripts should be required.
2. Navigate to 01-access-modeling-for-fuzzing/pw-discovery/
3. Run the experiments on the hosts by executing the ssh_based_kickoff_experiments.sh script
4. The experiments are spawning tmux sessions on the remote machines, so *tmux list-sessions* should provide a status on running experiments.
5. Collect the results from the fuzzing runs after the experiments have been finished (10 repetitions of 24 hour runs). This is done via the ssh_based_collect_results.sh script. The fuzzers shut themselves down automatically, so the experiment does not have to be cancelled manually. You may run the script at any time to collect intermediate results, but for the final result it is best to wait until the fuzzer has shut itself down. You may check whether the experiment is still running by checking the corresponding tmux session. Expect the experiments to run for 10-11 days including trace and per-run metrics generation.
6. Compute summary metrics via the run_metric_aggregation.py script.

For the P2IM unit tests of experiment (1), you may run 01-access-modeling-for-fuzzing/p2im-unittests/run_experiment.sh and observe the stdout output.

(2) State-of-the-art comparison. For the second experiment, the following workflow can be used to reproduce the experiments:

1. Same as for experiment (1).
2. Navigate to 02-comparison-with-state-of-the-art
3. Same as for experiment (1).
4. Same as for experiment (1).
5. Same as for experiment (1), but with 5 repetitions, 5-6 days of runtime, and using ssh_based_collect_results.sh.
6. Same as for experiment (1).

Note that experiments (1) and (2) are meant to be run in parallel in case 41 hosts are present, as experiment (1) is pre-configured to use 20 instances, while experiment (2) is pre-configured to use the remaining 21 instances.

(3) CVE discovery. For the third experiment, we tested the targets in large-scale fuzzing campaigns. As such, single 24-hour runs for replication do not make sense in this context. Instead, we provide crashing proof-of-concept inputs which were all generated in fuzzing runs, alongside with README's giving context on each POC. An example of this is 03-fuzzing-new-targets/zephyr-os/prebuilt_samples/CVE-2020-10065/POC/ within the fuzzware-experiments repository. You can still run the different CVE targets using the fuzzware pipeline utility (please refer to *fuzzware pipeline -h* for more documentation). We built each target such that previously known bugs are fixed (e.g., bugs of related CVEs), and crashing inputs generated via fuzzing should have a high likelihood to trigger the CVE bug.

(4) Crash Analysis. For the fourth experiment, we provide crashing POC inputs alongside some documentation on each input. The experiment's README at 04-crash-analysis/README.md contains an overview of how POC inputs correspond to the previous experiments, and how they can be run in Fuzzware.

A note on the multi-host setups. The base setup for (1) and (2) expect that the experiments are run in a distributed fashion on multiple hosts. In case your hardware resources do not allow for this multi-host setup, it is also possible to perform the same experiments on a smaller number of hosts that have access to more CPU cores. We provide scripts to run the experiments locally in the form of *run_experiment.sh* scripts within the respective experiment directories. As we cannot predict the exact computation resources (one very large host, a handful of medium-sized hosts, ...), these scripts are configured to run without parallelization by default. This means without modification, simply running the different *run_experiment.sh* scripts will take nearly a year to complete. However, we built parallelization and target specification options via environment variables into these scripts, such that the *run_experiments.sh* scripts should aid you in running the experiments according to a given hardware environment. Please refer to the script documentation for information on how to parallelize running the experiments within a host.

A.6 Evaluation and expected results

The main claims of the paper are:

1. Fuzzware employs a lightweight MMIO modeling technique.
2. Fuzzware's MMIO models reduce the fuzzer's input space considerably.
3. Fuzzware's MMIO models are applicable to a wide variety of firmware and hardware platforms.
4. Fuzzware outperforms the state-of-the-art.
5. Fuzzware's is able to identify previously unknown bugs.

The key results reported in the evaluation of the paper which support our claims are as follows:

1. Fuzzware's model generation cost an average of 6.34 minutes over 24-hour runs (6 seconds per model) for the pw-discovery data set.
2. On the same data set, Fuzzware achieves a minimum input elimination of 49.3% and a maximum 83.4%.
3. Fuzzware passes all of the valid P2IM unit tests.
4. In terms of basic block coverage, Fuzzware achieves on average 44% more coverage compared to P2IM and 61% more compared to uEmu.
5. Fuzzware's fuzzing campaigns yielded multiple previously unknown bugs in Zephyr-OS and Contiki-NG.
6. The majority of crashing inputs found by Fuzzware are true positives.

These claims are supported by the data generated when following the experiments in Section A.5. Note that the README file in each experiment sub-directory should provide additional context on what data is collected, where to find it, and what the expected results are.

After running the experiments, you should have access to a set of fuzzware-project directories that contain aggregated data. As an example, for the ARCH_PRO target of experiment (1), a directory *fuzzware-project-run-01* inside 01-access-modeling-for-fuzzing/pw-discovery/ARCH_PRO/ should have been automatically created. Similarly, for the P2IM/PLC sample of experiment (2), *fuzzware-project-run-01* should be present in 02-comparison-with-state-of-the-art/P2IM/PLC/. Running the *run_metric_aggregation.py* scripts should now output data in a similar representation to what can be found in the paper with regards to claim 1–4[1].

For claim 5–6, you may replay the given POC inputs and verify emulation behavior. In case you fuzz-tested the CVE targets with sufficient computation resources, you can also manually analyze the crashes which are produced in the respective fuzzware-project directories. We further include empiric timings for the first occurrence of according crashes in our experiments in 03-fuzzing-new-targets/README.md, alongside numbers on how many cores we used for the crash reproduction. Information on the reported bugs can be found in 03-fuzzing-new-targets/bug-details.

A.7 Experiment customization

In case your computation resources differ from our recommended setup, then modifications to the run scripts may be required to achieve experiment parallelization which matches your available setup. Please refer to the scripts' sources and README's for more information.

A.8 Notes

Due to the probabilistic nature of fuzzing, many of the numbers will differ in each run.

Furthermore, our basic block coverage collection is slightly different from the way it is collected in original publications for uEmu and P2IM. These papers report QEMUs translated blocks as reached basic blocks. However, due to the intrinsics of this emulator, these do not necessarily correspond to actual basic blocks. In our experiment, we match the entry of translated blocks to a list of actual basic blocks. While we include these allow lists in the repositories, you can generate them on your own by:

1. Opening the target's ELF file in IDA
2. While loading the binary, choosing ARMv7-M as the processor option
3. Running scripts/idapython/idapy_dump_valid_basic_block_list.py which is included in the fuzzware repository.
4. Execution function *dump_bbl_starts_txt()*.

You should now find a *valid_basic_blocks.txt* file next to the opened ELF file.

A.9 Version

Based on the LaTeX template for Artifact Evaluation V20220119.

[1] Note that we only include an automated experiment setup for Fuzzware, and not for rehosting frameworks we compare against.