USENIX'23 Artifact Appendix: ARMore: Pushing Love Back Into Binaries

Luca Di Bartolomeo
EPFL

Hossein Moghaddas
EPFL

Mathias Payer
EPFL

A Artifact Appendix

A.1 Abstract

ARMore 's artifact contains the source code necessary to run our static rewriter. This document describes how to set-up our prototype, gives a brief overview of the resource requirements to replicate some of the experiments conducted in our evaluation, along with instructions to run them.

A.2 Description & Requirements

This artifact is shipped as an aarch64 Docker container. The provided script `run.sh` takes care of importing the container and spawning a shell. All the relevant files for the experiment are in the root home folder.

A.2.1 Security, privacy, and ethical concerns

This artifact does not contain any threat to the system's integrity or privacy. However, we still recommend running it inside a sandboxed environment (either a VM or a container).

A.2.2 How to access

ARMore's artifact is available online at https://zenodo.org/record/7707936. ARMore's source code can be found at https://github.com/hexhive/retrowrite. Evaluators can visit commit 4a7193b to reproduce experiments shown in the paper.

A.2.3 Hardware dependencies

The evaluation of ARMore requires an aarch64 machine with large amounts of RAM (around 64 GB for all the experiments). Since we do not expect the evaluators to have access to such hardware, we provide in this artifact a reduced version of our experiments that should run even on an emulated aarch64 VM running on a x86 host with 8 GB of RAM. We provide instruction to run the experiments on both aarch64 or x86.

Note: if the evaluators consider running the full suite of experiments a necessity, we can provide remote ssh access to the required hardware.

A.2.4 Software dependencies

Since the artifact is shipped as a Docker container, all dependencies are already installed. However, in case evaluators would like to run it outside Docker, those are the required dependencies:

ARMore's evaluation was run on Ubuntu 20.04.2. The requried Ubuntu packages are `python3-pip`, `tcl-dev`, `build-essential`, `make`. The required python libraries are the following:

- archinfo
- pyelftools
- capstone (version >= 4.0.2)

the can be all installed by running from the home folder of the Docker container:
`pip3 install -r retrowrite/requirements.txt`
Finally, a working installation of `AFL++` is required.

A.2.5 Benchmarks

ARMore's evaluation in the paper used 3 different benchmarks:

- SPEC CPU 2017: to measure the baseline overhead introduced by ARMore, we use the entire SPEC benchmark. However, this benchmark requires large amount of RAM (64 GB) and considerable compute time (around 12 hours). For this reason, in this artifact evaluation we provide a reduced experiment using small benchmarks taken from https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
- MAGMA: to measure the overhead introduced by ARMore's coverage instrumentation, we run the MAGMA fuzzing benchmark. As before, this benchmark is quite expensive to build and run - we provide another minified version of this experiment to be able to run it in an emulated environment.
- SQLite test suite: we include the source code of the testsuite in the artifact along with scripts to test and run it.

A.3 Set-up

A.3.1 Installation

The artifact is shipped as an aarch64 docker image. Depending on the available hardware, we provide two options:

arm64 host: This is the preferred way, as it will make the experiments considerably faster. All the dependencies are already setup inside the Docker container. If not using the container, the following steps need to be taken: The following ubuntu packages need to be installed: `build-essential`, `python3-pip`, `tcl-dev`, `make`. Afterwards, go inside the `retrowrite` directory and run:
`pip3 install -r requirements.txt`
to install ARMore's dependencies.

x86 host: To run it on an x86 host, install the support for emulation of the arm64 architecture in docker images. The simplest way to do this is to run the following:
`docker run -privileged -rm tonistiigi/binfmt -install arm64`
as explained in https://docs.docker.com/build/building/multi-platform/. To test if multi-architecture support is running, you can try the following:
`docker run -rm arm64v8/alpine uname -a`.
Afterwards, download the artifact and use the `run.sh` script to import the image and spawn a shell inside the container.

Note: Experiment (E3) can only be run on an bare-metal arm64 host (i.e., not emulated). We found inconsistencies when running it through qemu-user (used by Docker).

A.3.2 Basic Test

To test basic functionality of ARMore, run `run.sh` to spawn a shell, go inside the home folder and run:

```
./retrowrite/retrowrite /bin/ls ls.s
./retrowrite/retrowrite -a ls.s rewritten_ls
./rewritten_ls
```

If the output is exactly the same as when running `/bin/ls`, then ARMore is set up correctly.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *ARMore's baseline overhead is <1%.* We provide evidence for this claim in experiment experiment (E1) which is a reduced version of the one described in Section 5.2 of the paper.

(C2): *ARMore's rewriting is correct and preserves functionality.* We provide evidence for this claim in experiment (E2), as described in Section 5.1 of the paper.

(C3): *ARMore enables efficient fuzzing of aarch64 closed-source binaries.* We provide evidence for this claim in experiment (E3), a reduced version of the one described in Section 5.4 of the paper.

A.4.2 Experiments

(E1): *[Baseline overhead] [10 human-minutes + 1 compute-hour]: This experiment shows how rewriting binaries without instrumentation adds negligible overhead (<1%). We took a random selection of binaries from the benchmarksgame* [1] *and use them to test the overhead introduced by ARMore.*

How to: Go inside the folder `~/claim_one_low_overhead`. The script `run.sh` will compile the benchmarks and store the binaries in the `compiled` folder. Afterwards, the script will rewrite the binaries with ARMore and store the result in the `rewritten` folder. Finally, the benchmarks will be run and the 2 different set of times will be printed.

Execution: Go inside the folder `~/claim_one_low_overhead` and run the script `run.sh`.

Results: While this experiment is certainly not conclusive compared to more heavy-weight benchmarks, the times noted by `Rewritten time` should be around 1% higher than the times noted by `Original time`.

(E2): *[Correctness] [5 human-minutes + 2 compute-hours]: This experiments verifies the correctness claims of ARMore denoted in Section 5.1, namely that it exactly preserves the original binaries' behaviour. This is done by rewriting the SQLite binaries and running their relevant testsuites.*

How to: Go inside the folder `~/claim_two_correctness`. The script `test_sqlite.sh` inside will build and rewrite the binaries from SQLite, and then run the testsuite on them.

Execution: Go inside the folder `~/claim_two_correctness` and run the script `test_sqlite.sh`, and check its output.

Results: The fifth to last line of the script output should report `0 errors out of 252696 tests`, indicating that all tests passed correctly.

(E3): *[coverage instrumentation overhead] [30 human-minutes + 2 compute-hours]: This experiment verifies the claims in Section 5.4, that is fuzzing with ARMore's coverage instrumentation is comparable to fuzzing with source-based instrumentation (`afl-cc`). Note: this experiment can only be run on a bare-metal aarch64 host.*

How to: Go inside the folder `~/claim_three_fuzzing`. The script inside will build the binaries from the first experiment (E1) and store the result in the `compiled` folder. Then, it will com-

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

pile them again with source instrumentation and store the result in the folder `source_instrumented`. Finally, the script will instrument the original binaries inside the `compiled` folder, adding coverage instrumentation, and store the result in the `rewritten_instrumented` folder.

Execution: Go inside the folder `claim_three_fuzzing` and run the script `run.sh` to build the instrumented binaries. Then, run the script `fuzz.sh` to fuzz each binary twice for 2 minutes: first, the source-instrumented version compiled with `afl-cc` will be fuzzed — secondly, the binary-instrumented version rewritten with ARMore will be fuzzed. The AFL UI is disabled, and only the executions per second are reported.

Results: As claimed in the paper, the average difference in executions per second should be slower for ARMore compared to `afl-cc` by around 25%. We note that this number is very variable, due to the non-deterministic nature of fuzzing.

If you get errors such as `/usr/bin/cat: output/default/fuzzer_stats: No such file or directory`, or the difference in executions per second is very large (e.g., 1000% instead of 25%), it means that this experiment was run inside an emulated Docker container instead of a bare-metal aarch64 host. We found this behaviour happening only inside qemu, but did not investigate fully the cause behind this. We suspect qemu's dynamic binary instrumentation and caching behavior on top of our static instrumentation may be the culprit. If the evaluators want access to an arm64 machine to run this experiment, please get back to us.

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/.