EL3XIR: Fuzzing COTS Secure Monitors

Christian Lindenmeier, *FAU Erlangen-Nürnberg;*
Mathias Payer and Marcel Busch, *EPFL*

https://www.usenix.org/conference/usenixsecurity24/presentation/lindenmeier

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

USENIX Security '24 Artifact Appendix: EL3XIR: Fuzzing COTS Secure Monitors

Christian Lindenmeier
FAU Erlangen-Nürnberg

Mathias Payer
EPFL

Marcel Busch
EPFL

A Artifact Appendix

A.1 Abstract

EL3XIR introduces a framework to effectively rehost and fuzz the secure monitor firmware layer of proprietary TrustZone-based TEEs. While other approaches have focused on naively rehosting or fuzzing Trusted Applications (EL0) or the TEE OS (EL1), EL3XIR targets the highly-privileged but unexplored secure monitor (EL3) and its unique challenges.

In our artifact evaluation, we demonstrate that state-of-the-art fuzzing approaches are insufficient to effectively fuzz COTS secure monitors. We give instructions on reproducing the results for EL3XIR's main technical contributions: (1) rehosting of secure monitor binaries, (2) interface recovery from REE OS, and (3) effectiveness of reflected peripheral modeling. We provide our evaluation set and instructions for executing fuzzing campaigns with EL3XIR. Additionally, we share details about assigned CVEs to support EL3XIR's capability to find unknown bugs.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our experiments are conducted within Docker containers and will not harm the host.

A.2.2 How to access

EL3XIR's source code can be found on Github at https://github.com/HexHive/EL3XIR/tree/ae-sec24-accepted. Since our evaluation set includes proprietary binaries, we provide all necessary evaluation data at Zenodo in an encrypted archive `el3xir-ae-part1-usenix24.tar.gz.enc`. We communicated the access key only with evaluators. Additionally, we provide the download link in EL3XIR's Github repository. The `README` in the archive gives an overview of all shared data. We note that for reproducing our results you have to collect your own artifacts (binaries, kernel source code).

A.2.3 Hardware dependencies

We conducted our experiments on a 16-core Intel Xeon Gold 5218 processor (hyperthreading disabled) with 64 GB of RAM. However, we expect that our results can be reproduced on machines with similar performance. Since we run the fuzzing campaigns (Section 6.3) limited to one CPU for each target eight times for 24 hours, we recommend using a processor with at least 8 cores. Additionally, we recommend at least 150 GB of free storage.

A.2.4 Software dependencies

We tested EL3XIR on Ubuntu 22.04. We require Git (tested `version 2.34.1`) to download the source code and repositories and Docker (tested `version 26.1.3, build b72abbb`) with the compose plugin (tested `version v2.27.1`) to build EL3XIR. Additionally, we need Python (tested `version 3.10.12` that comes with Ubuntu 22.04).

A.2.5 Benchmarks

EL3XIR requires at least the binary file of a targeted secure monitor for reproducing the fuzzing results without interface awareness ($iface^{-}/mmio^{-}$ and $iface^{-}/mmio^{+}$). For harness synthesis and reproducing the interface-aware fuzzing campaigns ($iface^{+}/mmio^{-}$ and $iface^{+}/mmio^{+}$), we require the source code of the deployed Linux kernel or the corresponding compiled LLVM-Bitcode files. Since our evaluation set includes proprietary binaries, we provide all necessary evaluation data at Zenodo in encrypted archives `el3xir-ae-part1-usenix24.tar.gz.enc`. We communicate the access key and instructions via HotCrp (see Artifact access) and links via EL3XIR's Github repository. Most importantly, the archive includes:

- Secure monitor binary: We compiled the open-source binaries from official sources and extracted the proprietary ones from firmware updates.
- Source code of REE OS kernel: For the open-source targets, we downloaded the Linux kernels from official sources. Considering the kernels deployed on COTS

Android smartphones, we directly provide the compiled LLVM-Bitcode files.

A.3 Set-up

A.3.1 Installation

We provide more details in the `README.md` file of EL3XIR's github repository. For installation with docker clone the repository (`git clone --recursive https://github.com/HexHive/EL3XIR.git`) and build the containers with 'make build'. In case you cannot build the images, we also provide pre-built docker images on Docker hub at https://hub.docker.com/r/chlindenmeier/el3xir_synthesis and https://hub.docker.com/r/chlindenmeier/el3xir.

To include our evaluation set, download the archives `el3xir-ae-part1-usenix24.tar.gz.enc`, decrypt it (see HotCrp), and move the content of the archives into EL3XIR's `in/` directory that will be mounted inside the docker containers. Each archive part includes artifacts for one target. We recommend to work with `el3xir-ae-part1-usenix24.tar.gz.enc`.

A.3.2 Basic Test

When the docker build process finished successfully, you can run 'make run-fuzz-sh' to spawn a shell in the container. For further testing, you can run a simple test fuzzing campaign ('$iface^{-}/mmio^{-}$'). To this end, check that the targeted secure monitor binary can be found at `/in/$TARGET/vendor-soc-bl31.bin`. We recommend to use our default target `TARGET=intel-n5x`. Then execute `make run-fuzz-test TARGET=$TARGET`. The test is successful, when EL3XIR boots the target, takes a snapshot, starts AFL++, and you can see AFL++'s dashboard.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The manual steps of EL3XIR's partial-rehosting approach are feasible (Section 6.1).

(C2): EL3XIR can perform automated interface recovery of function signatures for runtime services and synthesize effective fuzzing harnesses (Section 6.2 and Table 2).

(C3): EL3XIR outperforms state-of-the-art methods for fuzzing TEE firmware in edge coverage (Section 6.3, Figure 6, and Table 4).

(C4): EL3XIR can emulate MMIO accesses during fuzzing using reflected peripheral modeling to overcome coverage walls (Section 6.3 and Table 3).

(C5): EL3XIR can uncover previously unknown bugs in secure monitor implementations (Section 6.4 and Table 5).

A.4.2 Experiments

We design four experiments (**E1-E4**) to confirm **C1-C4**. For **C5** we provide an overview of CVE references as part of the downloaded archives `el3xir-ae-part1-usenix24.tar.gz.enc`. Read the `README` file in the archives for more information.

EL3XIR's experiments can mostly be reproduced with automated make targets. Look into EL3XIR's README file for more information. Note that estimate times provided assume the execution of the experiment for *one* target. We recommend to use our default target `TARGET=intel-n5x`. Furthermore, we note that since fuzzing is a non-deterministic process, some results may slightly differ from the exact numbers stated in our paper.

(E1): *[Rehosting Feasibility] [1 human-hour + 5 compute-minutes]:*

How to: EL3XIR's rehosting framework can be found at `src/rehosting-framework`. Each secure monitor requires the implementation of a rehosting environment (folder `secmonRehosting/rehostingEnvironments/`).
We manually inspect the complexity of a rehosting environment and boot the secure monitor binary to confirm **C1**.

Preparation: Open files `factories.py` and `$TARGET_boot_patcher.py`. You may inspect these files for multiple targets to confirm the systematical structure and similarity.

Execution: Manually inspect `factories.py` to find implementations of functions for software stubs (Section 6.1) including minimal bootloader, secure world stub, and normal world stub. Additionally, inspect `$TARGET_boot_patcher.py` to identify breakpoints and MMIO emulation to handle hardware dependencies. Run `make run-fuzz-test TARGET=$TARGET` to boot a targeted secure monitor binary. You may exit the docker container when AFL++'s dashboard comes up, since a snapshot has been taken.

Results: The objective is to boot the secure monitor until exit to normal world is reached, which is our attack surface for our fuzzer. Confirm that `out/$TARGET/snapshot.qcow2` is present. Inspect `out/$TARGET/qemu_err.txt` which holds the complete qemu logging for the boot process. Validate that the last instruction executed (end of file) was a `smc`, and that an exception return from EL3 to EL1 NS happened.

(E2): *[Interface Recovery] [1 human-hour + 30 compute-minutes]:*

How to: EL3XIR's interface recovery is mostly automated. We automatically synthesize an interface-aware fuzzing harness for an open-source target (Table 1) and reason about the results manually to validate **C2**.

Preparation: Manually create and fill file

in/$TARGET/llvm-link.lst with promising REE OS kernel object files that may exercise the targeted secure monitor's interface. You may also open our provided files, retrace our manual process, and validate the complexity.

Execution: Execute `make run-synth-eval TARGET=$TARGET` that will run the entire harness synthesis pipeline. This includes (1) compilation and linkage of the manually picked REE OS partition, (2) static analysis and interface recovery, and (3) probing of the SMC interface of the targeted secure monitor. You may ignore warnings during kernel compilation.

Results: Find the synthesized harness at `in/$TARGET/harnessdata.csv`. Each recovered interface is structured as described in Appendix B in our paper. To validate the correctness of interface prototypes (Table 2), compare the auto-generated harnessdata with our provided annotated harnessdata ground-truth file (`harnessdata_annotated.csv`). Furthermore, look into `/out/$TARGET/synth-summary-$TARGET.txt` to find a summarized report and compare it to Table 2.

(E3): *[Fuzzing] [10 human-minutes + 24 compute-hours]:*

How to: We conduct two fuzzing campaigns for one target using EL3XIR's baseline configuration ($iface^{-}/mmio^{-}$) representing the state-of-the-art and EL3XIR's full configuration ($iface^{+}/mmio^{+}$). We compare the resulting edge coverage to reason about **C3**.

Preparation: Ensure that secure monitor binary and `harnessdata.csv` are in `in/$TARGET`.

Execution: Execute `make run-fuzz-comp-eval TARGET=$TARGET` to start both fuzzing runs for ($iface^{-}/mmio^{-}$) and ($iface^{+}/mmio^{+}$). EL3XIR will automatically detect the number of CPU cores and assign half to each campaign while leaving leaving two free. If necessary you may set `NCORES` to a fixed number in the `Makefile` manually. The fuzzing campaigns will run 24 hours, but you may also halt them earlier using `make halt` (shortly wait until containers shut down) if necessary (e.g., after 5 hours). When finished run `make run-cov-comp-eval TARGET=$TARGET` to rerun all found test cases. You can plot an edge coverage graph for easy comparison using `make run-cov-comp-plot TARGET=$TARGET`. Warning and error message may be ignored if `out/$TARGET/plot.pdf` is generated.

Results: Inspect `out/$TARGET/plot.pdf` to see that EL3XIR's full configuration achieves higher edge coverage and results are similar to Figure 6 and Table 4 of our paper.

(E4): *[Reflected Peripheral Modeling] [10 human-minutes + 5 compute-minutes]:*

How to: We collect data about EL3XIR's reflect peripheral modeling by rerunning test cases with improved MMIO logging. The results confirm **C4**.

Preparation: We rerun test cases found during **E3**. Alternatively you may start a fresh fuzzing campaign with `make run-fuzz-eval TARGET=$TARGET HARNESS=$HARNESS MMIO_FUZZ=mmio`.

Execution: Execute `make run-mmio-comp-eval TARGET=$TARGET` to rerun all test cases found during **E3**'s fuzzing campaign with EL3XIR's full configuration ($iface^{+}/mmio^{+}$). If you started a fresh campaign, run `make run-mmio-eval TARGET=$TARGET HARNESS=$HARNESS MMIO_FUZZ=mmio`.

Results: Find a summary in file `mmio-summary-$TARGET-$HARNESS-mmio.txt` and compare the numbers to Table 3 of our paper to confirm that they look similar.

A.5 Notes on Reusability

EL3XIR is designed to be extended with other secure monitor binaries. For this purpose, collect necessary artifacts and place them in a new folder in `in/`. If trying to fuzz a new secure monitor with EL3XIR the following steps are required:

1. Implement rehosting environment: This includes a bootloader stub, secure world stub, normal world stup, and handling of hardware interactions. You may look into `../secmonRehosting/rehostingEnvironments/` to find examples. Section 4.1 of our paper explains a high-level systematic process for rehosting secure monitor binaries.

2. Optional harness synthesis: If you want to use EL3XIR's full potential you need the associated REE OS source code files (or already compiled as LLVM-Bitcode). Then write a file `llvm-link.lst` that defines all kernel object files that may be interesting for exercising the secure monitor's interface. Section 5.3 of our paper provides more details. Finally, run `make run-synth-eval` to automatically synthesize the harness.

3. Fuzzing: When you successfully rehosted the new secure monitor binary you may already fuzz it with EL3XIR without interface awareness but optionally with reflected peripheral modeling (`make run-fuzz-test TARGET=$TARGET HARNESS=noiface MMIO_FUZZ=$MMIO_FUZZ`). When harness synthesis is complete, you can also activate interface awareness.

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/usenixsec2024/.