USENIX Security '24 Artifact Appendix: "FFXE: Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries"

Ryan Tsang
University of California, Davis

Asmita
University of California, Davis

Doreen Joseph
University of California, Davis

Soheil Salehi
University of Arizona

Prasant Mohapatra
University of California, Davis

Houman Homayoun
University of California, Davis

A Artifact Appendix

A.1 Abstract

This appendix describes the software artifacts for the paper *FFXE: Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries*. It contains information on the contents of the artifact repository, the software requirements for usage, as well as instructions for setup and reproduction of experiments.

A.2 Description & Requirements

Artifacts were developed natively on an Apple Macbook M1 Pro, primarily in Python; however, to the best of our knowledge, none of primary the artifacts for reproduction are machine or architecture-specific. So long as the prequisite Python packages install without error, tests for FFXE itself should be able to run natively (this is untested on Windows).

We have provided a Dockerfile that also installs additional software for running tests for the other tools FFXE was compared against. The build time for the Dockerfile is quite long, however, and upon request, we can provide a prebuilt docker image file directly. *The Dockerfile has not been tested on Windows.*

Please note that because we could not obtain approval from healthetile.io to release their firmware artifacts, the case study outlined in the main paper under Section 5.4 is unavailable for evaluation.

A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, executing the code in our repository should not present any risk to the machine security of any evaluators.

It should be noted that we do run the Docker container provided with `-privileged` enabled due to performance limitations in unprivileged mode.

A.2.2 How to access

Our artifact repository at commit `17adcd8` can be accessed on GitHub at the following link: https://github.com/rchtsang/ffxe/tree/17adcd8

A.2.3 Hardware dependencies

Our artifacts have been tested on an Apple Macbook M1 Pro with 16GB RAM and 8 performance cores.

Multiple cores should not be necessary for evaluation. We have not tested the software on a machine with less RAM within recent history and cannot make guarantees for machines with less than that; however, we believe less RAM (around 10-12GB) should not present a major issue.

No other specific hardware features should be necessary.

A.2.4 Software dependencies

Software was tested natively and in a Docker container backed by a vanilla Colima instance.

To run only FFXE tests natively, the following software is needed:

- `conda/mamba >= 23.9.0` (recommend `miniforge`)

Required Python packages are managed by `conda` environments whose YAML files can be found in the repository under `docker/envs`.

To run tests for other tools for comparison purposes, the following additional software is needed:

- `Ghidra >= 10.3.2`
- `Ghidrathon >= 4.0.0`

We will not provide installation instructions for Ghidra and Ghidrathon as we have provided a Dockerfile should the evaluator be interested in running those tests.

Note that for full reproduction of our real-world tests, we also rely on FirmXRay to locate base addresses before beginning our analysis. As this is an older project, it is difficult to install natively and we highly recommend using the provided

Dockerfile, though this should not be necessary for recreation of immediate results.

To run tests in Docker, the following software is needed:

- `Docker >= 25.0.2`
- `make >= 3.81` (convenience Makefile on Mac/Linux to build/run docker container)

A.2.5 Benchmarks

No addtional datasets need to be fetched for this artifact, as all necessary samples are already included in the repository.

A.3 Set-up

A.3.1 Installation

Native (for running FFXE tests only)

Once `conda/mamba` has been installed, the environment for running FFXE tests can be installed by running the following command from the root directory:

```
conda env create -f docker/envs/ffxe.yml
```

ffxe must be installed with the following command at the project root to make use of scripts:

```
pip install -e .
```

Docker

Once `docker` is installed, a docker daemon must be running before invoking any docker commands. If Docker for Desktop is installed, the application should be open to begin running the daemon. Colima is another alternative and can be initiated with `colima start`.

Ensure that the daemon is live by running `docker version`. If no message saying "is docker daemon running?" is displayed, then the daemon is live.

To build and run the docker container, use the following commands:

```
# build docker image
$ docker build -t ffxe/workspace:dev docker

# start a container
$ docker run \
    -t -d                   \
    --privileged            \
    --init                  \
    --name=ffxe-workspace   \
    --entrypoint=bash       \
    -v `pwd`:/home/ffxe     \
    ffxe/workspace:dev

# get a shell in container
$ docker exec -w /home/ffxe -it ffxe-workspace
 ↪ /bin/bash

# kill running container
$ docker rm -f ffxe-workspace
```

A.3.2 Basic Test

Native (for FFXE tests only)

The basic FFXE tests can be invoked by running the following from the repository root:

```
# activate conda environment
$ conda activate ffxe
# install ffxe as package (must be done on
 ↪  first activation of ffxe env)
$ pip install -e .
# run the basic tests
$ python tests/test-unit.py
```

A series of results should be printed for the basic firmware samples. If this is the case, the real-world samples can also be run using `scripts/test-real.py`.

Running both of these will produce CFGs that be used in later comparisons to reproduce results.

Docker

The same tests as above can also be performed in docker after getting a shell in a built container.

However, do note that due to limitations of docker on MacOS, the performance is considerably worse. These limitations will not be present if run in a docker container on Linux.

A.4 Evaluation workflow

A.4.1 Major Claims

We make the following major claim in our paper:

(C1): *FFXE is able to resolve CFG edges corresponding to indirect function calls for registered functions in interrupt service routines where other tools cannot.*

Other claims made in our paper are incidental to this major claim. (Our claims regarding complementary coverage comparison follow directly from this major claim. Real-world comparisons did not explicitly test for this claim and were included at the request of the reviewers during revisions. Those comparisons can be repeated, but we will not outline the steps here. All code used to produce our main papers figures is present in our artifact repository.)

A.4.2 Experiments

(E1): *Resolving Registered Functions [30 human-minutes + <=2 compute-hour + <1GB disk]: In this experiment, multiple tools including FFXE are used to recover CFGs from a set of known firmware samples taken from the Nordic NRF5 SDK which contain indirect calls to registered functions. Recovered CFGs are then analyzed to check if any edges to registered function sites are*

actually resolved and the results are tabulated into the LaTeX-formatted table used for Table 1 of the main paper.

How to: To reproduce this experiment, run scripts to perform CFG extraction for sdk samples for each tested engine: FXE, FFXE, Ghidra, and `angr`. The extracted CFGs will then be compared against the known registered function locations found in `tests/registered-functions.json` to check if any edges to those locations were successfully identified.

Preparation: Scripts for each set of tools requires activating a different conda environment, which can be done as follows from the root directory:

```
$ conda activate ffxe ; pip install -e .
$ conda activate ghidra
$ conda activate angr
```

Environments should be deactivated before activating a new one.

Execution:

1. Run FFXE tests (natively or in docker)

```
$ # conda env must be active
$ [ $(basename $CONDA_PREFIX) = "ffxe"
↪  ] || conda activate ffxe
$ # ffxe must be installed in conda env
↪  via pip
$ [ pip show ffxe &> /dev/null ] || pip
↪  install -e .
$ python tests/test-unit.py
$ conda deactivate
```

2. Run FXE tests (natively or in docker)

```
$ # conda activate ffxe
$ python tests/fxe-all.py
$ # conda deactivate
```

3. Run ghidra tests (natively or in docker)

```
$ # conda activate ghidra
$ python scripts/ghidra-analyze.py
$ # conda deactivate
```

4. Run angr tests (natively or in docker)

```
$ # conda activate angr
$ python scripts/angr-analyze.py
$ # conda deactivate
```

Results: The script to generate the Table 1 from our main paper analyzes each CFG and looks for edges to the known indirectly-called functions listed in `tests/registered-functions.json`.

The script will print out the latex-formatted table and can be run after all above scripts are run using the commands:

```
$ conda activate ffxe
$ python scripts/tabulate-regfuncs.py
```

A.5 Notes on Reusability

FFXE currently outputs its CFG as a python pickle file serialized with the `dill` package. The output CFG is represented essentially as a dictionary containing the keys "nodes" and "edges".

The "nodes" value is a set of tuples containing the start address of the basic block and its size in bytes.

The "edges" value is a set of tuples containing a "from" address and a "to" address that represent the beginning and end of an edge.

Obviously this isn't a terribly useful format for direct analysis, but it was chosen to make comparisons presented in the paper easier. There is a better CFG and basic block model classes in the `models.py` file.

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/.