Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software

Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth, *University of Lübeck*

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

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

USENIX'23 Artifact Appendix: Cipherfix: Mitigating Ciphertext Side-Channel Attacks in Software

Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth
University of Lübeck, Lübeck, Germany
{j.wichelmann, a.paetschke, l.wilke, thomas.eisenbarth}@uni-luebeck.de

A Artifact Appendix

A.1 Abstract

CIPHERFIX is a framework for finding and mitigating ciphertext side-channel leakages in software. It combines dynamic binary instrumentation and dynamic taint tracking to pinpoint vulnerable code parts. Then, it hardens the binaries against ciphertext side-channel leakage with the help of static binary instrumentation.

This artifact comprises our source code and usage instructions. We offer prebuilt Docker images which contain all necessary dependencies, library binaries and precompiled examples. The GitHub repository presents detailed instructions on how to build, run and extend CIPHERFIX.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our source code is available at `https://github.com/UzL-ITS/Cipherfix`. The commit used to reproduce the results in the paper is `0d05fcb`. The artifact contains our dynamic analysis (`static-variables`, `structure-analysis` and `taint-tracking`), the static mitigation (`static-instrumentation`) and our evaluation modules (`memwrite-tracer` and `evaluation`).

A.2.3 Hardware dependencies

For running CIPHERFIX, an AMD Zen1/Zen2/Zen3 CPU is highly recommended (we tested on an AMD EPYC 7763 and on an AMD EPYC 3151). Other x86 CPUs may also work, but CIPHERFIX does not have support for all instructions (e.g., AVX-512), so there might be unsupported instructions and therefore potential instabilities.

A.2.4 Software dependencies

We offer precompiled Docker images which contain all necessary dependencies.

For building the whole framework and the examples from scratch without Docker, please refer to the Prerequisites and Compiling sections in the README.

A.2.5 Benchmarks

The example targets for evaluating the performance and security of CIPHERFIX are located in the `examples` directory. The `cipherfix-examples-full` Docker image contains precompiled binaries for all examples.

A.3 Set-up

A.3.1 Installation

As we ship the artifact as a precompiled Docker image, only a Docker installation is required.

Our precompiled image (`ghcr.io/uzl-its/cipherfix-examples-full`) was built on Zen3, which helps reproducibility on other systems. For example, compilers may check for certain CPU features and then emit instructions which are not yet supported by our instrumentation framework.

If you want to rebuild the Docker images, follow these steps:

1. Clone the CIPHERFIX repository.
2. Run `./build-docker-images.sh` to build the Docker images. You may need `sudo`, if your local user is not member of the system's `docker` group.

A.3.2 Basic Test

Pull and run our precompiled Docker image:

```
docker run -it \
  ghcr.io/uzl-its/cipherfix-examples-full
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Dynamic Analysis

(Section 4.2.1) The static variable detection tool writes information about static variables in the program into a `static-vars.out` file.

(Sections 4.1, 4.2) The `static-vars.out` file is subsequently read by the taint tracking, which adds information about detected heap allocations and stack frames. The taint analysis then tracks which memory location contain secrets, and marks instructions that accessing them. All taint analysis results are written into a `taint.out` file.

(Section 4.3) To aid static instrumentation, a structure analysis tool collects basic blocks and register/flag liveness information. The results are written into a `structure.out` file.

The aforementioned files are acquired by experiment (E1).

(C2): Static Instrumentation

(Section 5) The static instrumentation tool reads the dynamic analysis results and generates hardened binaries, as shown by experiment (E2). The hardened binaries are functionally correct.

(C3): Functional Correctness and Overhead

(Section 6.2) The hardened binaries are functionally correct, but slower than the original ones. This is verified in experiment (E3).

A.4.2 Experiments

In the following, we describe how to verify the claims made in the previous section. For this, we have prepared a number of scripts that run CIPHERFIX for two representative targets.

For the specific steps, we refer to the Running the example targets section in the README file.

(E1): [Dynamic Analysis] [3 human-minutes + 0 compute-hours + 50 MB disk]:

How to: Follow steps 1 and 2 of the Running the example targets section in the README file.

The taint analysis may print a number of warnings and a few errors about unknown instructions. Usually, those can be safely ignored.

Results: The result files end up in `/cipherfix/cipherfix/examples/mbedtls/aes-multiround/` respectively `/cipherfix/cipherfix/examples/wolfssl/eddsa/`.

(E2): [Static Instrumentation] [10 human-minutes + 0 compute-hours + 15 MB disk]:

How to: Follow step 3 of the Running the example targets section in the README file.

Note that the `structure.out` files need to be manually extended with information about heap allocations functions, as described in the README.

Results: The static instrumentation was successful when there are several `.instr` files in the `/cipherfix/cipherfix/examples/mbedtls/aes-multiround/instr-fast-aesrng` and `/cipherfix/cipherfix/examples/wolfssl/eddsa/instr-fast-aesrng` directories.

(E3): [Functional Correctness and Overhead] [3 human minute + 0 compute-hours]:

How to: Follow step 4 of the Running the example targets section in the README file.

Results: The standard output shows the computed ciphertexts and signatures for both the original and the hardened binaries. If the outputs are identical, the functional correctness is given. The `Loop time` specifies the time needed for the cryptographic computations and allows computing the overhead.

A.5 Notes on Reusability

Our proof-of-concept implementation includes modules for the dynamic analysis and the static instrumentation module, as well as evaluation modules. Each of them has a fixed input and output format, which is documented in the `docs` folder in the repository. As long as these formats are followed, each module can be replaced without modification of the other components. For example, it is possible to use another dynamic analysis engine, or an alternative binary rewriting framework with better performance guarantees. See the Replacing Framework Modules section in the README for more information.

A.6 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/`.