USENIX Security '25 Artifact Appendix: Efficient Ranking, Order Statistics, and Sorting under CKKS

Federico Mazzone
University of Twente

Maarten Everts
University of Twente
Linksight

Florian Hahn
University of Twente

Andreas Peter
Carl von Ossietzky
Universität Oldenburg

A Artifact Appendix

A.1 Abstract

The artifact consists of the complete codebase supporting our paper, which introduces a novel approach to performing ranking, order statistics, and sorting under CKKS, a fully homomorphic encryption scheme. Specifically, the artifact includes a C++ implementation of the four functionalities discussed in the paper: ranking, argmin, median, and sorting, all built on top of the OpenFHE library for CKKS.

This implementation was designed to benchmark the runtime of these functionalities, the primary empirical result of our paper, across vectors of increasing length in both single- and multi-threaded settings.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We do not identify any security risks associated with installing or using our artifact.

A.2.2 How to access

We have made our artifact available on multiple platforms:

- As permanent storage on Zenodo: https://doi.org/10.5281/zenodo.14673904
- As working repository on Github: https://github.com/FedericoMazzone/openfhe-statistics
- As docker container on Docker Hub: https://hub.docker.com/r/mazzonef/openfhe-statistics

A.2.3 Hardware dependencies

No specific hardware is required, as our library runs exclusively on CPU. OpenFHE and CKKS require significant RAM, particularly for larger experiments (see Table 2 in our paper). In the largest experiment (sorting with 16,384 elements), memory consumption may reach up to 230 GB.

To replicate our performance results, we used a machine equipped with an Intel Xeon Platinum 8358 processor running at 2.60 GHz, featuring 32 cores (64 threads) and 512 GB of RAM.

A.2.4 Software dependencies

Our implementation relies on the OpenFHE library, available at https://github.com/openfheorg/openfhe-development. We specifically use version 1.1.2, which is included in our repository to avoid compatibility issues.

Building and running our implementation requires a C++ compiler and CMake. We recommend using GCC v10 or Clang 11 on Linux for best compatibility.

We have tested our repository exclusively on Linux Ubuntu. While it should theoretically compile and function on Microsoft Windows, we do not guarantee compatibility.

A.2.5 Benchmarks

We use synthetic data points generated in the range [0,1] with a precision of two decimal places. The exact dataset is available in the file `data/points1d.csv`.

A.2.6 Benchmarks

We use synthetic data points generated in the range [0,1] with a precision of two decimal places. The exact dataset is available in the file `data/points1d.csv`.

A.3 Set-up

A.3.1 Installation

Unless using the Docker container, both the OpenFHE library and our library must be compiled. Installation steps are detailed in the README file, but a summary is provided below:

1. Install prerequisites:
   (a) `sudo apt-get install build-essential`
   (b) `sudo apt-get install cmake`
2. Install OpenFHE:

(a) `cd openfhe-development-1.1.2`
(b) `mkdir build`
(c) `cd build`
(d) `cmake ..`
(e) `make -j`
(f) `sudo make install`
(g) `cd ../..`

3. Install our library:
   (a) `mkdir build`
   (b) `cd build`
   (c) `cmake ..`
   (d) `make -j`
   (e) `cd ..`

If you do not have sudo access in your machine, specify a different installation path for OpenFHE. See more about this in the README file.

A.3.2 Basic Test

To run a basic test, execute `./build/demo`. This runs a demonstration of all four functionalities on a fixed vector. The expected console output includes initialization of the CKKS cryptosystem, followed by the execution of ranking, argmin, median, and sorting in sequence.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our four functionalities (ranking, argmin, median, and sorting) exhibit asymptotic quadratic runtime, achieving the runtimes reported in Figure 6 and memory consumption reported in Table 2 of the paper. This is validated by the experiment (E1) described in Section 6 of the paper.

A.4.2 Experiments

(E1): Main experiment, 1 human-minute + 717 compute-hour + 230GB memory
Preparation: Compile and install the library.
Execution: Run the command `sh ./benchmark.sh <algorithm>`, where `algorithm` is one of the following: ranking, ranking-tie, minimum, median, sorting. The script will automatically execute tests on vectors of increasing length (from 8 to 16,384) in both single- and multi-threaded settings.
Results: Runtime and memory consumption results are stored in the CSV file `benchmark.out` (one row per experiment). On a comparable machine, performance metrics (runtime and memory) should be similar to those reported in the paper. Note that running the Docker version may introduce runtime overhead, particularly for short vectors.

A.5 Notes on Reusability

The parameters of our tests are set to work with data with two decimal digits precision. In case the input data needs to be more or less precise, the comparison approximation parameters need to be adjusted accordingly. See more about this in the README 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/usenixsec2025/.