

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Bridging Usability and Performance: A Tensor Compiler for Autovectorizing Homomorphic Encryption

Edward Chen, Fraser Brown, and Wenting Zheng, *Carnegie Mellon University*

https://www.usenix.org/conference/usenixsecurity26/presentation/chen-edward

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology



USENIX Security '26 Artifact Appendix: Bridging Usability and Performance: A Tensor Compiler for Autovectorizing Homomorphic Encryption

Edward Chen
Carnegie Mellon University
ejchen@cmu.edu

Fraser Brown
Carnegie Mellon University
fraserb@cmu.edu

Wenting Zheng
Carnegie Mellon University
wenting@cmu.edu

A Artifact Appendix

A.1 Abstract

This artifact contains Rotom, a compilation framework that autovectorizes tensor programs into optimized homomorphic encryption (HE) programs. The artifact includes the complete Rotom compiler implementation (approximately 12,000 lines of Python), a comprehensive suite of tensor workloads from HE compiler literature and state-of-the-art secure inference protocols, automated scripts to reproduce experiments from the paper, and documentation with setup instructions, usage examples, and guides for extending Rotom. The artifact supports the paper's main claims by enabling reviewers to compile tensor programs and verify Rotom's compilation times (Figure 6), execute compiled HE programs and measure runtime performance (Figure 7), reproduce microbenchmark results demonstrating ApplyRoll optimizations (Tables 1-3), and verify Rotom's performance against baseline compilers (Fhelipe [2] and Viaduct-HE [3]). All experiments target the CKKS implementation in OpenFHE [1] and can be reproduced on a standard Linux machine with sufficient CPU cores and memory.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns associated with evaluating this artifact. All experiments run in user space within a Python virtual environment. The artifact performs purely computational tasks (tensor compilation and homomorphic encryption operations) on synthetic benchmark data. No real user data, personally identifiable information, or sensitive information is processed. The OpenFHE library used for executing compiled programs is a well-established, open-source homomorphic encryption library with no known security vulnerabilities that would affect artifact evaluation.

A.2.2 How to access

The artifact is publicly available through three channels. For reproducibility and artifact evaluation, we provide a permanent archived version on Zenodo at `https://zenodo.org/records/17957733`, which corresponds to the version of Rotom evaluated for this paper. For convenience, we provide a pre-built Docker image at `https://ghcr.io/edwjchen/ae-rotom:latest`, which includes all dependencies and baseline compilers pre-installed. For the latest updates, bug fixes, and ongoing development, the Rotom repository is available on GitHub at `https://github.com/cmu-cryptosystems/Rotom`.

A.2.3 Hardware dependencies

Rotom does not require any specialized hardware and can run on standard commodity CPUs. However, for optimal performance and to reproduce the paper's results, we recommend a multi-core processor with at least 16 cores (64 cores recommended for full benchmarks), 256 GB RAM required for larger benchmarks, and at least 10 GB of free disk space. The evaluation in the paper was conducted on a 64-core Intel Xeon CPU with 256 GB of RAM.

A.2.4 Software dependencies

Rotom was tested on Ubuntu 24.04 LTS. The artifact should work on other modern Linux distributions with minimal modifications. The core software dependencies are Python 3.12, NumPy for numerical operations, and OpenFHE for executing compiled homomorphic encryption programs. All Python dependencies, including specific versions of NumPy and other required packages, are specified in the `requirements.txt` file included in the artifact and can be installed automatically using `pip`. OpenFHE is an open-source fully homomorphic encryption library available at `https://github.com/openfheorg/openfhe-development`, and installation instructions are provided in the artifact documentation.

Docker (recommended): To simplify dependency installation and baseline setup, we provide a pre-built Docker image.

Pull and run the image with:

```
docker pull ghcr.io/edwjchen/ae-rotom:latest
docker run -it ae-rotom bash
```

The working directory inside the container is `/app/Rotom`.

A.2.5 Benchmarks

The benchmark suite consists of six application-level tensor workloads and three microbenchmarks, all of which are included directly in the artifact. All benchmarks use synthetic data.

Application-level benchmarks: (1) MatMul performs ciphertext-plaintext matrix multiplication, (2) Double-MatMul performs two consecutive ciphertext-ciphertext matrix multiplications, (3) TTM computes the third-order tensor matrix product, (4) Convolution performs 2D convolution over multi-channel images, (5) LogReg-MatVecMul performs two consecutive matrix-vector multiplications for logistic regression, and (6) BERT Attention implements the attention layer from the BERT-base transformer model.

Microbenchmarks: The artifact includes three targeted microbenchmarks that evaluate the performance of ApplyRoll and its optimized variants (ApplyRotRoll, ApplyBSGSRoll, ApplyBSGSMatMul) compared to ApplyPermute, as well as the effectiveness of Strassen's algorithm.

Baseline compiler benchmarks: To compare Rotom's compile time against Fhelipe and Viaduct-HE, the artifact requires both Fhelipe and Viaduct-HE. These repositories can be found through https://github.com/fhelipe-compiler/fhelipe and https://github.com/rolph-recto/viaduct-he/.

The artifact provides another directory containing Rotom's benchmarks that can eb copied into each of these systems. This directory contains instruction traces for various benchmark operations from both baseline systems. Some traces have been hand-modified to ensure compatibility with Rotom's wrapper interface and to enable execution on Rotom's OpenFHE backend. Installation instructions for these benchmarks are provided in the directory. An online version of these benchmarks can also be found here: https://github.com/edwjchen/rotom_benchmarks.

A.3 Set-up

Option A: Docker (recommended). Pull and launch the prebuilt image:

```
docker pull ghcr.io/edwjchen/ae-rotom:latest
docker run -it ae-rotom bash
```

Inside the container, run the provided setup script to clone and copy benchmarks and scripts into the correct locations:

```
/app/setup.sh
```

The working directory is `/app/Rotom`. Proceed directly to the Basic Test below.

Option B: Manual installation. Clone the Rotom repository, create a Python virtual environment, and install dependencies using `pip install -r requirements.txt`.

A.3.1 Basic Test

Run `pytest` in the Rotom directory to execute all unit tests. Run `python main.py` to compile and execute a basic matrix multiplication benchmark.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Rotom achieves fast compilation times, compiling all tensor workloads in under 5 minutes (most in seconds). This is proven by experiment (E1) described in Section 8.2.1, whose results are reported in Figure 6.

(C2): Rotom achieves high search quality, matching or outperforming state-of-the-art HE compilers (Fhelipe and Viaduct-HE) by up to 80× in execution time. This is proven by experiment (E2) described in Section 8.2.2, whose results are reported in Figure 7.

(C3): Rotom's ApplyRoll operator provides significant performance improvements over ApplyPermute for layout conversions, achieving 3-8× speedup in common scenarios. This is proven by experiments (E3) described in Section 8.1, whose results are reported in Tables 1-3.

A.4.2 Experiments

(E1): *[Compilation Time] [30 human-minutes + 48 compute-hours]: Measure Rotom's compilation time for all benchmarks and compare against Fhelipe and Viaduct-HE to validate claim (C1).*

Preparation: If using Docker, run `/app/setup.sh` inside the container; this clones `rotom_benchmarks`, Fhelipe, and Viaduct-HE and copies all benchmarks and scripts into the correct locations automatically. If building manually, ensure all dependencies are installed for Rotom by following its README, then build Fhelipe (https://github.com/fhelipe-compiler/fhelipe) and Viaduct-HE (https://github.com/rolph-recto/viaduct-he/) following their respective instructions. Copy `rotom_benchmarks/fhelipe_apps` into `fhelipe/frontend/fheapps/rotom` and `rotom_benchmarks/viaduct_apps` into `viaduct-he/benchmarks`.

Execution: Run `python run_all_tests.py` to compile (and execute) all benchmarks with Rotom, Fhelipe, and Viaduct-HE at both $n = 8K$ and $n = 32K$. The script automatically records compilation times of Rotom and

saves results to `Rotom/logs`. To compare against Fhelipe and Viaduct-he, we provide compilation scripts within `rotom_benchmarks/scripts` and they are intended to be run within the top-level directory of each system. Note that each of the bert-attention benchmarks takes around 30 minutes to an hour to run.

Results: Use `python parse_logs.py` to parse all of Rotom's logs and generate a plot of Rotom's compile time.

(E2): *[Runtime Performance] [10 human-minutes + 48 compute-hours]: Measure execution time of compiled HE programs from Rotom, Fhelipe, and Viaduct-HE to validate claim (C2).*

Preparation: If using Docker, run `/app/setup.sh` inside the container; as stated in (E1). If building manually, first ensure that all instruction traces are within `Rotom/benchmarks`. These traces are found in https://github.com/edwjchen/rotom_benchmarks. Both `fhelipe_benchmarks` and `viaduct_benchmarks` need to be copied to `Rotom/benchmarks`. Additionally, ensure all Python dependencies are installed. Please run `pytest` to execute all unit tests for Rotom.

Execution: Run `python run_all_tests.py` to execute all compiled programs on the OpenFHE backend. The script runs each benchmark 5 times and records average execution time, communication costs (LAN/WAN), and operation counts. Results are saved to `Rotom/logs`. Note that each of the bert-attention benchmarks takes around 30 minutes to an hour to run.

Results: After `python run_all_tests.py` has finished, use `python parse_logs.py` to parse all of Rotom's logs. This script should generate both compile and execution times for Rotom (and other systems) as a plot in `Rotom/plots`.

(E3): *[Microbenchmarks] [5 human-minutes + 10 compute-minutes]: Evaluate ApplyRoll optimizations and compare against ApplyPermute to validate claim (C3).*

Preparation: No additional preparation required beyond basic installation.

Execution: Run `./scripts/rotom/benchmark_roll_conversion.sh`, `./scripts/rotom/benchmark_slot_roll_conversion.sh` and `./scripts/rotom/benchmark_strassens.sh` to execute three microbenchmarks: (1) swapping two dimensions, (2) swapping two slot dimensions with BSGS, and (3) Strassen's algorithm with different conversions. Results are saved to `Rotom/logs/*`, where * is the name of the script (without `benchmark_`).

Results: Compare operation counts (Add, Mul, Rot) and execution times in the respective logs.

Note: Compute time estimates assume a 64-core machine with 256 GB RAM. Machines with fewer resources may require proportionally longer execution times, particularly for larger benchmarks.

A.5 Notes on Reusability

Rotom is designed to be extensible and reusable for a variety of homomorphic encryption research and applications beyond the experiments in this paper.

Writing custom tensor programs: Users can write new tensor programs using Rotom's DSL, which closely resembles PyTorch. The `frontends/tensor.py` module provides common operators (MatMul, Conv2D, Add, etc.) that can be composed to build custom workloads. Examples of tensor programs are provided in `benchmarks/` and can serve as templates for new applications.

Integrating with other HE backends: While Rotom currently targets OpenFHE, it can be extended to support other HE libraries by implementing a new backend in `backends/`. The toy backend (`backends/toy.py`) serves as a plaintext example for correctness testing and can used as a reference for other frameworks.

Visualizing layouts: Rotom includes a layout visualization tool (`layout_visualizer.py`) that helps understand how tensors are packed into ciphertexts, useful for debugging and educational purposes.

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/usenixsec2026/.

References

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, 2022.

[2] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proceedings of the ACM on Programming Languages*, 8(PLDI):126–150, 2024.

[3] Rolph Recto and Andrew C Myers. A compiler from array programs to vectorized homomorphic encryption. *arXiv preprint arXiv:2311.06142*, 2023.