USENIX Security '26 Artifact Appendix: Libra: Pattern-Scheduling Co-Optimization for Cross-Scheme FHE Code Generation over GPGPU

Song Bian
Beihang University

Yintai Sun
Beihang University

Zian Zhao
Beihang University

Haowen Pan
Beihang University

Mingzhe Zhang*
Unaffiliated

Zhenyu Guan*
Beihang University

A Artifact Appendix

A.1 Abstract

Libra is an end-to-end fully homomorphic encryption (FHE) compiler for GPGPUs that transforms high-level C programs into efficient GPU FHE implementations. Libra automates efficient code generation by co-optimizing cross-scheme computational patterns with hardware-aware scheduling strategies. This artifact includes: (i) the Libra compiler source code, (ii) the underlying FHE libraries (based on FlyHE), (iii) comprehensive benchmarks (microbenchmarks and end-to-end applications), and (iv) scripts that automate compilation, execution, result collection, and figure/table generation to reproduce the paper's key results.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact raises no security, privacy, or ethical concerns. It performs standard compilation and benchmarking on the included workloads and does not require disabling security mechanisms or handling sensitive data. All evaluations use only synthetic or publicly available benchmark datasets. The workflow is self-contained and does not permanently modify the host system configuration. The only external interaction is optionally pulling a public Docker image.

A.2.2 How to access

The artifact is publicly available through the following resources:
- **Source Repository (GitHub):** https://github.com/sunnchioo/Libra
- **Docker Image:** https://hub.docker.com/r/suen0/libra_ae
- **Archived Snapshot (Zenodo):** https://zenodo.org/records/17962002

*Mingzhe Zhang and Zhenyu Guan are corresponding authors.

A.2.3 Hardware dependencies

To reproduce the performance results reported in the paper, we recommend the same GPU platform used in the evaluation:
- **GPU:** NVIDIA A100 GPGPU ($\geq$ 40 GB).
- **CPU Architecture:** AMD64.
- **Host Memory:** $\geq$ 80 GB.

Other NVIDIA GPUs may run parts of the pipeline but may not reproduce performance numbers comparable to the paper.

A.2.4 Software dependencies

The artifact has been tested on Ubuntu 22.04. We provide a Docker image to ensure environment consistency.
Docker (Pre-installed): The image includes all toolchains and heavily-compiled dependencies (e.g., Polygeist, LLVM, and MLIR) pre-installed.
Manual Build Requirements:
- CMake $\geq$ 3.31.1; Ninja $\geq$ 1.10.1
- GCC/G++ $\geq$ 13.1.0; Clang/Clang++ $\geq$ 22.0.0
- CUDA Toolkit $\geq$ 12.4
- LLVM & MLIR $\geq$ 22.0
- NTL $\geq$ 11.5.1; GMP $\geq$ 6.2.1
- Python $\geq$ 3.10 (matplotlib $\geq$ 3.10.6, pandas $\geq$ 2.3.3)

A.2.5 Benchmarks

All benchmarks used in the paper are included under `Libra_full_bench`, consisting of:
- **Microbenchmarks:** Individual FHE operators.
- **Applications:** End-to-end workloads.

A.3 Set-up

A.3.1 Installation

We provide two options for installing Libra.

Option A: Docker (Recommended) We recommend using Docker to avoid dependency and toolchain mismatches.
1. Pull the Docker image:

```
docker pull suen0/libra_ae
```

2. Launch the container with GPU access:

```
sudo docker run --gpus all -it \
  --name libra_compiler \
  --memory=80g --memory-swap=84g \
  suen0/libra_ae:v1.0 /bin/bash
```

3. The source code, data, and required dependencies for the artifact are available in `Security_Artifact/Libra`.

Option B: Build from Source If not using Docker, install all dependencies listed above and build the toolchain components from the repository root (`Security_Artifact/Libra`) in the following order.

1. Build Polygeist:

```
cd Libra/Tool/Polygeist
mkdir -p build && cd build
cmake -G Ninja ../llvm-project/llvm \
  -DLLVM_ENABLE_PROJECTS="clang;mlir" \
  -DLLVM_EXTERNAL_PROJECTS="polygeist" \
  -DLLVM_EXTERNAL_POLYGEIST_SOURCE_DIR=.. \
  -DLLVM_TARGETS_TO_BUILD="host" \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_BUILD_TYPE=Release
ninja -j32
ninja check-polygeist-opt
ninja check-cgeist
```

2. Build LLVM/MLIR/Clang:

```
cd Libra/Tool/llvm-project
mkdir -p build && cd build
cmake -G Ninja ../llvm \
  -DLLVM_ENABLE_PROJECTS="mlir;clang" \
  -DLLVM_TARGETS_TO_BUILD="host" \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_BUILD_TYPE=Release
ninja -j32
ninja check-mlir
```

3. Build Libra compiler:

```
cd Libra
mkdir -p build && cd build
cmake -G Ninja ..
ninja
```

A.3.2 Basic Test

This basic test verifies that the frontend portion of the pipeline runs and produces intermediate outputs.

```
cd Libra
./Script/libra-mlir-base.sh
./Script/libra-opt-base.sh
./Script/libra-translate-base.sh
```

Expected outcome: The script completes without errors and generates base CUDA program files under the `MLIR` subdirectory in `Libra_full_bench/`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) Microbenchmark performance: Libra generates FHE kernels that outperform or match the provided reproducible baselines on microbenchmarks. This corresponds to results shown in Figure 8.
(C2) End-to-end application performance: Libra achieves significant speedups on application workloads compared to state-of-the-art baselines. These correspond to results shown in Figures 9–10, Table 4 and the ResNet-20 results.
(C3) Performance breakdown: Libra's speedups are explained by its co-optimization of cross-scheme computational patterns and hardware-aware scheduling. This corresponds to the performance breakdown shown in Figure 11.

A.4.2 Experiments

(E1): Full evaluation pipeline (reproduces C1–C3)
[2 human-hours + 3–5 compute-hours + 40GB disk]
Execution:
1. Generate CUDA code from the benchmark C programs:

```
cd Libra
./Script/libra-mlir.sh
./Script/libra-opt.sh
./Script/libra-translate.sh
python Script/libra-trans.py
```

2. Compile the foundational FHE libraries:

```
cd Libra/HElib/FlyHE
mkdir -p build && cd build
cmake ..
make -j
```

3. Run benchmarks and baselines:

```
cd Libra
./Libra_plot/run_evaluation.sh
```

To improve transparency, these intermediate IR files are intentionally preserved, enabling users to inspect the internal compiler passes, optimization stages, and cost models.
Results Collection:
Raw results are saved to:

```
Libra_plot/output/output.txt
```

To regenerate the plots and tables reported in the paper:

```
cd Libra
python ./Libra_plot/plot_all.py
```

A.5 Notes on Reusability

The artifact is designed to be modular and reusable beyond the paper. Libra is built on MLIR with a clear backend abstraction interface. Users can inspect the intermediate IR transitions to understand the cost models, and extend dialects and passes (e.g., in `Libra/Libra/Dialect` and `Libra/Libra/Tools`) to support new FHE schemes, patterns, or targets.

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/.