USENIX Security '26 Artifact Appendix: Orbit: Optimizing Rescale and Bootstrap Placement with Integer Linear Programming Techniques for Secure Inference

Zikai Zhou*† William Seo*‡ Edward Chen‡ Alex Ozdemir§ Fraser Brown‡ Wenting Zheng‡
†Tsinghua University ‡Carnegie Mellon University §Max Planck Institute for Security and Privacy
zhouzk22@mails.tsinghua.edu.cn, {wys, ejchen, fraserb, wenting}@cmu.edu, alex.ozdemir@mpi-sp.org

A Artifact Appendix

A.1 Abstract

This artifact contains Orbit, an FHE compiler that jointly optimizes bootstrap and rescale placement for CKKS secure-inference workloads through a level-scale-aware integer linear programming (ILP) formulation. It includes the Orbit implementation (about 3,500 lines of Python and 2,500 of Go), MLIR inputs for five CNNs (ResNet-20, SqueezeNet, AlexNet, MobileNet, VGG16) in SiLU and ReLU variants, profiled Lattigo cost models, and scripts that reproduce the paper's experiments. The artifact packages Orbit itself; the baseline placement algorithms (DaCapo [3], Orion [4], ReSBM [6]) are not included, so it reproduces Orbit's own results and compares them against the baseline numbers reported in the paper. Reviewers can compile every benchmark and check Orbit's compilation times (Tables 6–7), estimate or execute the compiled programs to check Orbit's latencies and operation counts (Figures 5–6, Table 8), and verify accuracy within 0.3% of plaintext PyTorch (Table 9). All experiments target CKKS in Lattigo on a standard Linux machine.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. All experiments are purely computational, run in user space, and use only synthetic inputs and public CIFAR-10 samples.

A.2.2 How to access

An archived version matching the evaluated paper is on Zenodo (`https://zenodo.org/records/20278435`); ongoing development is on GitHub (`https://github.com/cmu-cryptosystems/Orbit`).

*Equal contribution.

A.2.3 Hardware dependencies

Orbit needs only a commodity CPU and under 5 GB of RAM to compile. To reproduce the paper we recommend a high core count (QBP solving is parallelized) and, for executing the larger benchmarks, up to 256 GB of RAM; the paper compiled on AWS `m8i.8xlarge` (32 cores) and executed on `m7g.16xlarge` (64 cores, 256 GB). Allow about 20 GB of disk for compilation and one benchmark's execution data; the plaintext constants for *all* benchmarks total order 50 GB.

A.2.4 Software dependencies

Orbit was tested with Python 3 (the Docker image pins Python 3.11 on Debian, with Go 1.24 for the backend). Python dependencies (`gurobipy`, `joblib`, `matplotlib`, `more-itertools`, `networkx`, `numpy`) are in `requirements.txt` and installable with `pip`. The `gurobipy` package ships a size-limited license sufficient for every standard (partitioned) benchmark; only the `-nopart` and `-nocomp` ablations exceed its cap and need a full (e.g., free academic) Gurobi license. The DaCapo frontend (generates plaintext inputs) and Lattigo backend (executes compiled MLIR) are needed only for execution; **reviewers checking only estimated results need neither.** See `frontend/README.md` and `backend/README.md`, or build both inside the container with `scripts/setup_dependencies.sh`.

A.2.5 Benchmarks

Five CNNs used by prior FHE compilers—ResNet-20, SqueezeNet, AlexNet, MobileNet, VGG16—each in a SiLU variant (single-Chebyshev SiLU, average-pooling) and a ReLU variant (three-Chebyshev ReLU, max-pooling; pooling only in AlexNet, SqueezeNet, VGG16). They are MLIR files in `mlirs_input/`, lowered by DaCapo into Orbit's FHE Computation IR (the paper's shared input representation). Accuracy is validated against plaintext PyTorch on CIFAR-10.

A.3 Set-up

A.3.1 Installation

Install the Python dependencies:

```
git clone \
  https://github.com/cmu-cryptosystems/Orbit
cd Orbit && pip install -r requirements.txt
```

To execute (not only estimate) programs, also install the frontend and backend (their READMEs), or use the Docker image below—which needs no manual Python or Gurobi setup.

A.3.2 Basic Test

Run `pytest`, then compile one benchmark end to end:

```
python3 scripts/optimizer/orbit/run_orbit.py \
  --model ResNet --act SiLU \
  --n 64 --Lm 16 --Sw 40
```

Output lands under `mlirs_output/`: the `.txt` log reports `Orbit Compilation time` and the estimated `Final tdag latency`; the `.mlir` is the compiled program.

A.3.3 Reproducing the experiments with Docker

Build the image once and drive it with `scripts/reproduce.sh`; the image bundles the dependencies and the size-limited Gurobi license, so the cost-model experiments (E1, E3) need no further setup:

```
docker build --build-arg USER_ID=$(id -u) \
    --build-arg GROUP_ID=$(id -g) -t orbit:dev .
docker run --rm orbit:dev   # tests+smoke
```

The experiments below run `reproduce.sh` targets through this image; mounting the repository with `-v $PWD:/opt/orbit` keeps generated files (compiled MLIRs and logs) on the host. The `-nopart`/`-nocomp` CompPart ablations additionally need a full Gurobi license, added with `-v $HOME/gurobi.lic:/opt/gurobi/gurobi.lic:ro` and `-e GRB_LICENSE_FILE=/opt/gurobi/gurobi.lic`. To execute (E2, E4), first build the backend and frontend and generate inputs once, then use the `execute` target:

```
D="docker run --rm -e HOME=/tmp \
   -v $PWD:/opt/orbit"
$D orbit:dev \
    ./scripts/setup_dependencies.sh backend
$D orbit:dev \
    ./scripts/setup_dependencies.sh frontend
$D -w /opt/orbit/frontend/dacapo orbit:dev \
    bash -c 'source .venv/bin/activate && \
    ../dacapo_patch/gen_all_mlirs.sh && \
    python3 examples/tests/gen_input_data.py 10'
$D orbit:dev ./scripts/reproduce.sh execute \
    --model ResNet --act SiLU \
    --n 16 --Lm 16 --Sw 40 --run 0
```

Decrypted output lands in `mlirs_execute/`, comparable to the PyTorch reference from the data-generation step (E4).

A.4 Evaluation workflow

A.4.1 Major Claims

The baselines (DaCapo, Orion, ReSBM) are **not** part of this artifact; each claim is reproduced for Orbit, and the paper computes the comparisons against the reported baseline numbers.

(C1): Orbit compiles every benchmark and configuration in under 6 minutes; the reproduced times are the Orbit columns of Tables 6–7. Verified by E1.

(C2): Orbit's joint bootstrap-and-rescale ILP lowers execution cost: its compiled programs use up to 5.4× fewer rescales and 31–34% fewer bootstraps than Orion (Figures 5–6, Table 8), yielding the paper's speedups of 7–19% over DaCapo, 2–73% over Orion, and up to 52% over ReSBM. Verified by E2.

(C3): Orbit's DAG-reduction techniques (compression, partitioning, QBP reuse, bypass handling) cut ILP solving time while keeping end-to-end latency within 1% (Tables 3–5). Verified by E3.

(C4): Orbit stays within 0.3% of plaintext PyTorch accuracy across all benchmarks (Table 9). Verified by E4.

A.4.2 Experiments

Each suite below is shown as its native script `auto_scripts/<name>.sh`; equivalently, run it through the Docker image as `reproduce.sh <t>`—short for `docker run -rm -v $PWD:/opt/orbit orbit:dev ./scripts/reproduce.sh <t>`—whose targets mirror the script names (e.g., `compile_orbit_eval_base.sh` ≡ `compile-base`). Execution (E2, E4) additionally needs `-e HOME=/tmp` and the one-time backend/frontend build from Set-up. Below, the instructions use the commands for compilation and execution in Docker.

(E1): Compilation time [15 human-minutes + several compute-hours]. Compile the four configuration suites (base n=64k, plus 16k, L_{bts}=12, and S_{min}=51):

```
reproduce.sh compile-base
reproduce.sh compile-16k
reproduce.sh compile-lm12
reproduce.sh compile-sw51
```

Each `.txt` log under `mlirs_output/` reports `Orbit Compilation time`: on the recommended multi-core machine every benchmark compiles in under 6 minutes, matching the Orbit columns of Tables 6–7. Compilation parallelism follows `ORBIT_THREADS` (default 32; set it to the machine's core count).

(E2): Runtime performance [10 human-minutes + ∼1-hour setup + minutes–hours per benchmark]. Install the frontend and backend (or use Docker) to execute, or skip for estimates only. Compile as in E1, then run each program through the `execute` target (the command shown in Set-up) on Lattigo; `compile-sim-vari` selects the variable-bootstrapping cost model for the ReSBM comparison. Compare Orbit's latencies and the rescale/bootstrap counts printed in each `.txt` log against the Orbit data in Figures 5–6 and Table 8. Building the frontend/backend and generating inputs is a one-time ∼1-hour step; plaintext evaluation then takes seconds to minutes (∼10 s for AlexNet-SiLU at 16k) and encrypted evaluation minutes to hours (∼25 min for AlexNet-SiLU at 16k; the 64k benchmarks run for hours and need ∼256 GB RAM).

(E3): DAG-reduction microbenchmarks [10 human-minutes + 1–2 compute-hours]. Run the ablation suites; `-nocomp`, `-nopart`, `-nobypass`, and `-qbp` on `run_orbit.py` toggle the individual techniques:

```
reproduce.sh compile-micro-comppart
reproduce.sh compile-micro-bypass
reproduce.sh compile-micro-reqbp
```

Compare compilation times, partition counts, and latencies against Table 3 (compression/partitioning), Table 4 (QBP reuse), and Table 5 (bypass); end-to-end latency stays within 1%. Each ablation compiles in seconds to ∼30 minutes. The `-nopart`/`-nocomp` variants in `compile-micro-comppart` are the slowest and exceed the bundled Gurobi license [5], so they require a full license (mounted as in Set-up); the other ablations run on the bundled license.

(E4): End-to-end accuracy [10 human-minutes + ∼1-hour setup + hours per benchmark]. Install the frontend and backend so CIFAR-10 inputs and PyTorch references are generated into `input_data/` (a one-time ∼1-hour build and data-generation step). Execute the compiled benchmarks at $n = 2^{17}$ on encrypted inputs and compare with the PyTorch outputs (Table 9): accuracy is within 0.3%, most models matching exactly (max difference order 10^{-4}). [1]

A.5 Notes on Reusability

Orbit is a modular pass over a backend-agnostic CKKS IR, so it retargets to other FHE libraries (e.g., SEAL, OpenFHE) via a new backend lowering and integrates into frameworks such as HEIR [1] and Rotom [2] through its MLIR interface. Compile arbitrary MLIR with `optimizer.py`; the core components—the ILP formulation (`ilp_core.py`), SISO partitioning and bypass handling (`siso_partition.py`), iterative solving and merging (`iterative_partition.py`), QBP management (`qbp_manager.py`), and compression (`auto_compression.py`)—are organized for extension, and new backend/hardware cost models drop into `cost_models/` as profiled JSON.

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/.

[1] The accuracy for $n = 2^{15}$ varies from PyTorch execution due to bugs in Dacapo frontend. As stated in the paper, this configuration is not intended for deployment.

References

[1] Asra Ali, Jaeho Choi, Bryant Gipson, Shruthi Gorantala, Jeremy Kun, Wouter Legiest, Lawrence Lim, Alexander Viand, Meron Zerihun Demissie, and Hongren Zheng. Heir: A universal compiler for homomorphic encryption. *arXiv preprint arXiv:2508.11095*, 2025.

[2] Edward Chen, Fraser Brown, and Wenting Zheng. Bridging usability and performance: A tensor compiler for autovectorizing homomorphic encryption. *Cryptology ePrint Archive*, 2025.

[3] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. {DaCapo}: Automatic bootstrapping management for efficient fully homomorphic encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6993–7010, 2024.

[4] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 734–749, 2025.

[5] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026.

[6] Yan Liu, Jianxin Lai, Long Li, Tianxiang Sui, Linjie Xiao, Peng Yuan, Xiaojing Zhang, Qing Zhu, Wenguang Chen, and Jingling Xue. Resbm: Region-based scale and minimal-level bootstrapping management for fhe via min-cut. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 924–939, 2025.