HECO: Fully Homomorphic Encryption Compiler
Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi, ETH Zurich
https://www.usenix.org/conference/usenixsecurity23/presentation/viand

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

Open access to the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.
HECO is a compiler for Fully Homomorphic Encryption built using the MLIR compiler framework. It translates imperative programs (defined in a high-level intermediate representation) into the SIMD-like paradigm required for most FHE schemes. It uses Microsoft SEAL as the underlying FHE implementation, generating C++ code that is then compiled and linked against the SEAL library. HECO uses xDSL (a Python-based "sidekick" to the C++ based MLIR framework) for its frontend, which features a simple embedded Domain Specific Language (DSL) that allows developers to specify FHE computations in a straight-forward manner.

The artifact also contains reference C++ implementations written directly against SEAL. For each of the programs we evaluate, we provide two implementations: one representing a "naive" non-expert baseline, and one "optimal" implementation based on the batching approaches generated by the synthesis-based Porcupine tool, which HECO is compared against in the paper.

HECO is available as open-source software at github.com/MarbleHE/HECO. The evaluated artifact, specifically, is available at github.com/MarbleHE/HECO/tree/artifact.

HECO does not have specific hardware requirements. Note, however, that the “naive” versions of some of the evaluation workloads require at least 10 GB of free memory.

HECO should be cloned using git (git clone https://github.com/MarbleHE/HECO). After cloning, it is necessary to initialize the git submodules that are used to provide HECO’s external dependencies, which are the LLVM/MLIR framework and the Microsoft SEAL library: git submodule update --init --recursive.

Before HECO can be built, the MLIR framework needs to be built. For evaluation, it is recommended to build MLIR in Release configuration. Note that compiling MLIR can require significant time, ranging from around 20 min on a powerful desktop or server, to up to two hours on weaker laptops. Assuming the current working directory is the HECO repository root, execute the following to build MLIR:

```bash
USENIX Association
Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium 271
```
mkdir dependencies/llvm-project/build
cd dependencies/llvm-project/build
cmake -G Ninja ../llvm \
-DLLVM_ENABLE_PROJECTS=mlir \ 
-DLLVM_BUILD_EXAMPLES=OFF \ 
-DLLVM_TARGETS_TO_BUILD=X86 \ 
-DLLVM_ENABLE_ASSERTIONS=ON \ 
-DLLVM_BUILD_TYPE=Release \ 
-DLLVM_CODEC_COMPILER=clang \ 
-DLLVM_CODEC_COMPILER=clang++ \ 
-DLLVM_ENABLE_LLD=ON \ 
-DLLVM_INSTALL_UTILS=ON \ 
-DMLIR_INCLUDE_INTEGRATION_TESTS=OFF

In order to compile and run the generated C++ code, the Microsoft SEAL library needs to be installed. This can be done (from the HECO repository root) as follows:
cd ../../seal 
cmake -S . -B build 
cmake -build build 
sudo cmake -install build
cd ../..

HECO, like its dependencies, uses the cmake build system. The dependencies are not automatically included in the HECO build structure (i.e., not added via add_subdirectory) and cmake will search for the dependencies during configuration. While SEAL’s installation will be automatically detected, MLIR requires providing a MLIR_DIR path. Assuming SEAL and MLIR have been built as indicated above, execute the following (from the repository root) to build HECO:
mkdir build 
cmake -S . -B build \
-DMLIR_DIR=dependencies/llvm-project\ 
/build/lib/cmake/mlir 
cmake -build build -target heco

A.3.2 Basic Test

After building HECO, you can call heco without any parameters and feed in any *.mlir file as input, which should round-trip and output the unmodified input program:
../build/bin/heco < test/example.mlir

You can also test the full compilation flow, by first compiling the High-Level Intermediate Representation (HIR) into a low-level C-friendly Intermediate Representation (emitC):
../build/bin/emitc -full-pass\ 
< test/example.mlir > test/out.mlir
This can then be translated into C/C++ source code:
../build/bin/emitc-translate -mlir-to-cpp\ 
< test/out.mlir > test/out.cpp

A.4 Evaluation workflow

A.4.1 Major Claims

The paper makes the following major claims about the artifact

(C1): HECO produces code that achieves significant speedup compared to naive/non-batched FHE implementations (i.e., up to several orders of magnitude faster). This is demonstrated by (E1) which compares the performance of naive implementations with HECO-produced code and is described in Section 6.1 of the paper, with results highlighted in Figure 5.

(C2): HECO produces code that matches the performance of “optimally”batched code. This is shown by the second half of (E1) which compares HECO to the synthesis-based Porcupine tool and is described in Section 6.2 of the paper, with results highlighted in Figure 6.

(C3): HECO’s solution scales to real-world problem sizes (which synthesis-based tools fail to do). This is shown by (E2), which shows HECO’s compile time for various problem sizes and is described in Section 6.1, with results shown in Table 1.

A.4.2 Experiments

In the following, we describe the experiments. Note that all time estimates assume the set-up process (which includes the potentially lengthy compilation of the LLVM/MLIR dependency) has been completed. Detailed instructions can also be found in evaluation/README.MD.

(E1): Speedup (evaluation/benchmark)

In order to reproduce the results of Figure 5, the inputs need to be first compiled using HECO and then run using the Microsoft SEAL library. In addition, the naive baseline implementations need to be run using SEAL, too. This requires the vast majority of the (compute-)time, as the naive baseline implementations quickly become significantly less efficient (i.e., over 15min for a single problem, compared to fractions of a second for the HECO optimized version). Running this experiment should require around 15 human-minutes and no more than 2 compute-hours.

Preparation: In order to compile the programs from the high-level intermediate representation (HIR) form given here to *.cpp, you can use a helper script that does this for all files in the heco_input folder (assuming your current working directory is the repository root):
../evaluation/benchmark/heco_helper.sh
You can then compile and build the benchmark target (assuming your current working directory is the repository root): cmake -build build -target benchmark

Execution: Execute the generated binary (.build/bin/benchmark). This will
create a set of *.csv files of the format `<workload>_HECO_<size>.csv` in evaluation/plotting/data/benchmark.

**Results:** The *.csv files report one iteration on each line, reporting key generation time, encryption time, evaluation time, and decryption time (in this order) in microseconds. In evaluation/plotting/plot_all.py a rough plotting script is provided, including a pipfile that defines the necessary dependencies. In addition, the plotting requires LaTeX to be installed.

(E2): **Comparison** (evaluation/comparison)

In order to reproduce the results of Figure 6, the procedure is similar to that for Experiment 1. However, in addition to the HECO versions and naive baseline implementations, there are also Porcupine reference implementations. As in the previous experiment, the naive baselines consume the vast majority of the compute time. Running this experiment should require around 15 human-minutes and no more than 2 compute-hours.

**Preparation:** In order to compile the programs, you can use a helper script that does this for all files in the heco_input folder (assuming your current working directory is the repository root):

```
./evaluation/comparison/heco_helper.sh
```

You can then compile and build the comparison target (assuming your current working directory is the repository root): cmake -build build -target comparison

**Execution:** Execute the generated binary (`./build/bin/comparison`). This will create a set of *.csv files of the format `<workload>_HECO_<size>.csv` in evaluation/plotting/data/comparison.

**Results:** The *.csv files report one iteration on each line, reporting key generation time, encryption time, evaluation time, and decryption time (in this order) in microseconds. In evaluation/plotting/plot_all.py a rough plotting script is provided, including a pipfile that defines the necessary dependencies. In addition, the plotting requires LaTeX to be installed.

(E3): **Compile Time** (evaluation/compile_time)

In order to reproduce the results of Table 1, the programs need to be compiled with the mlir-timing option. Running this experiment should require around 30 human-minutes.

**Preparation:** No additional preparation is required.

**Execution:** Compile each of the provided HIR inputs with the timing flags `-mlir-timing -mlir-timing-display=list` added. You can use a helper script that does this for all files in the heco_input folder (assuming your current working directory is the repository root):

```
./evaluation/compile_time/heco_helper.sh
```

**Results:** The execution time report includes the Total Execution Time, which can be compared to the results in the paper. Note that a significant fraction of the compile time is usually spent in the Canonicalizer pass, which is a built-in pass from MLIR. As a result, compile times might vary significantly as MLIR updates and changes the underlying framework. Please also note that MLIR must be built in Release configuration in order to achieve acceptable compile time performance.

A.5 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/.