USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FABLE: Batched Evaluation on Confidential Lookup Tables in 2PC

Zhengyuan Su, *Tsinghua University;* Qi Pang, Simon Beyzerov, and Wenting Zheng, *Carnegie Mellon University*

https://www.usenix.org/conference/usenixsecurity25/presentation/su-zhengyuan

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

USENIX Security '25 Artifact Appendix: FABLE: Batched Evaluation on Confidential Lookup Tables in 2PC

Zhengyuan Su*, Qi Pang†, Simon Beyzerov†, Wenting Zheng†
*Tsinghua University, †Carnegie Mellon University

A Artifact Appendix

A.1 Abstract

This artifact provides the code and instructions to reproduce the major claims of FABLE, an efficient 2PC protocol for confidential LUT evaluation, with lightweight client computation, concretely efficient server computation, scalable communication, and LUT confidentiality. In particular, it provides the instructions and scripts to reproduce all evaluation results of FABLE and its baselines. It also contains easy-to-use scripts to reproduce FABLE's advantages in the two applications mentioned in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, executing this artifact does not impose any risk on the machine's security, data privacy, or any other ethical concern. The artifact showcases an efficient 2PC protocol that helps enhance user privacy in real-world collaborative computation. We believe the artifact is safe to execute.

A.2.2 How to access

The artifact is available at https://doi.org/10.5281/zenodo.15586635, which contains two git repositories.

If you prefer to download from GitHub, please clone the repositories https://github.com/cmu-cryptosystems/FABLE and https://github.com/cmu-cryptosystems/FABLE-AE, and place them side by side under the same folder.

A.2.3 Hardware dependencies

Our code does not require specific hardware features to compile and run. However, to help reproduce the numbers shown on the paper, we provide *two c6i.16xlarge instances on AWS EC2*.

We would like to provide the reviewers with access via SSH to our two `c6i.16xlarge` instances, which contain the environments to execute the code. To gain access to the machines, please follow the instructions on the HotCRP system.

A.2.4 Software dependencies

In our artifact, we provide multiple Dockerfiles to set up the environments.

To set up the environment to build and run FABLE, build the Dockerfiles under the two folders:

```
docker build ./FABLE -t fable:1.0
docker build ./FABLE-AE -t fable-ae:1.0
```

Note that you may need to run the commands under sudo if you are not a root user.

Some baselines, including the DORAM baselines and Crypten's application baseline, require separate environments to execute. However, they are not mandatory because we provide their performance numbers in the artifact. If you prefer to reproduce those numbers, please run the following commands to build the Docker images:

```
bash FABLE-AE/ORAMs/build-dockers.sh
docker build FABLE-AE/Crypten -t crypten
```

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

FABLE is installed inside the Docker environment while the image `fable:1.0` is built. Therefore, no operation is needed for installation other than building the Dockerfile.

A.3.2 Basic Test

To check that the installation is successful, first start a Docker container with

```
docker run -it -w /workspace/FABLE fable:1.0
```

Inside the container, run the following commands:

```
./build/bin/fable 127.0.0.1 r=1 l=1 &
./build/bin/fable 127.0.0.1 r=2 l=1
```

It will run a toy example of FABLE in 10 seconds, report the breakdown of computation and communication, and check the result.

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims of FABLE consist of the following:

(C1): FABLE achieves orders of magnitude speedup over prior baselines under both LAN and WAN networks. In particular, FABLE achieves up to 28-101× speedup in LAN environments and up to 50-393× speedup in WAN environments. This is proven by the end-to-end performance evaluation described in Section 6.3 and Figure 5.

(C2): FABLE's performance benefits from multi-threading and batched processing. The benefit from multi-threading is described in Section 6.2 and Figure 4. The benefit from batched processing is described in Section 6.3, Figure 6, and Figure 7.

(C3): FABLE's performance remains competitive even when the OPRF is replaced with the more secure AES, as shown in Table 4.

(C4): FABLE also exhibits orders of magnitude speedup under two applications. As explained in Section 6.4, FABLE achieves a speedup of 456× (178×) for secure embedding lookup and a speedup of 15× (27×) for secure query execution under LAN (WAN).

A.4.2 Experiments

(E1): [Main Experiments] [30 human-minutes + 10 compute-hour + 1GB disk]: run FABLE under various configurations and the SPLUT baseline, and reproduce the main plots.

Preparation: In this experiment, we use the `fable-ae` image. Specifically, in the root directory of our artifact, launch a container with

```
docker run -it --net=host --cap-add=NET_ADMIN
    -v $PWD/FABLE-AE:/workspace/AE -w
    /workspace/AE fable-ae:1.0
```

on two instances. Then, set two environment variables on both instances: `HOST` and `CLIENT`, which are the IP addresses of the server and the client, respectively.

Execution: To launch the main experiments, run

```
bash main_experiments.sh 1 0.0.0.0
```

and

```
bash main_experiments.sh 2 $HOST
```

on the server and the client, respectively.
After the main experiments finish, please launch the SPLUT experiments with

```
bash baseline_splut.sh 1 0.0.0.0
```

and

```
bash baseline_splut.sh 2 $HOST
```

on the server and the client, respectively.
Then, synchronize the experiment results with

```
bash sync_log_recv.sh $CLIENT
```

and

```
bash sync_log_serv.sh
```

on the server and the client, respectively.
Finally, produce the plots and the tables with

```
python3 microbench.py
python3 draw.py
python3 breakdown_table.py
python3 breakdown_table.py --aes
```

on the server.

Results: You should find 60 log files under `FABLE-AE/logs/main` and 12 log files under `FABLE-AE/logs/baseline/splut` on both instances. On the server side, you will also find the following plots under `FABLE-AE/plots`:
Figure 4: `runtime_vs_threads.pdf`.
Figure 5: `time_lutsize_network_clipped.pdf`.
Figure 6: `24_time_bs.pdf`.
Figure 7: `comm.pdf`.
Also, the execution of `breakdown_table.py` prints Table 4 in the standard output.

(E2): [Application] [30 human-minutes + 1.5 compute-hour]: run the applications and reproduce the speedups.

Preparation: In this experiment, we still use the `fable-ae` image. Specifically, in the root directory of our artifact, launch a container with

```
docker run -it --net=host --cap-add=NET_ADMIN
    -v $PWD/FABLE-AE:/workspace/AE -w
    /workspace/AE fable-ae:1.0
```

on two instances. Then, set two environment variables on both instances: `HOST` and `CLIENT`, which are the IP addresses of the server and the client, respectively.

Execution: To launch the application experiments, run

```
bash applications.sh 1 0.0.0.0
```

and

```
bash applications.sh 2 $HOST
```

on the server and the client, respectively.
Afterwards, run `python3 read_app_speedup.py` to see the speedup for both applications.

Results: You should find 6 log files under `FABLE-AE/logs/applications` on both instances.

The Python file will print the speedups of both applications in the standard output, which should be over 100× for the embedding lookup and over 10× for the secure join execution.

(E3): [Optional Reproduction of DORAM Baselines] [30 human-minutes + 1.5 compute-hour]: reproduce the DORAM baseline performance. This step is *optional* as we already provide the evaluation results in our artifact.
Preparation: In this experiment, we use DUORAM's image. Please ensure that you have already executed

```
bash FABLE-AE/ORAMs/build-dockers.sh
```

Execution: To launch the DORAM baseline experiments, run

```
bash FABLE-AE/ORAMs/repro-dockers.sh
```

on the server.
Afterwards, run `python3 draw.py --doram-baseline` to see the plots with the DORAM baseline logs.
Results: You should find 12 log files under `FABLE-AE/logs/baseline/floram` and another 12 log files under `FABLE-AE/logs/baseline/duoram`. You will find the same set of plots as E1.

(E4): [Optional Reproduction of the Crypten baselines] [30 human-minutes + 1.5 compute-hour]: reproduce the Crypten baseline performance. This step is also *optional* as we already provide the evaluation results in our artifact.
Preparation: In this experiment, we use the `crypten` image. Please launch the Docker container on two instances:

```
docker run -it --net=host --cap-add=NET_ADMIN
    -v $PWD/FABLE-AE:/workspace/AE -w
    /workspace/AE crypten
```

Then, set two environment variables on both instances: `HOST` and `CLIENT`, which are the IP addresses of the server and the client, respectively.
Execution: To launch the experiments, run

```
bash Crypten/reproduce.sh 0 $HOST $CLIENT
```

and

```
bash Crypten/reproduce.sh 1 $HOST $CLIENT
```

on the server and the client, respectively.
Afterwards, run

```
python3 read_app_speedup.py --crypten-baseline
```

to see the speedup for both applications.
Results: You should find 10 log files in total under `FABLE-AE/logs/applications`. The speedups will be similar to E2.

A.5 Notes on Reusability

The reproduction only uses a subset of all flags supported in FABLE. In this section, we elaborate the flags of the executable `build/bin/fable`.

- `p`: the port number for communication. Default: 8000.
- `bs`: the batch size of the input. Default: 4096.
- `db`: the LUT size. Default: `2^LUT_INPUT_SIZE`, i.e., the maximum supported size.
- `seed`: the random seed. Default: 12345.
- `par`: Whether enable parallelization. Default: 1 (i.e., enable parallelization).
- `thr`: Number of threads used in parallelization. Only useful when `par=1`, Default: 16.
- `l`: Type of the LUT. It will not affect the performance of FABLE. See the README for more information. Default: 0 (i.e., Random LUT).
- `h`: The OPRF type. Default: 0 (i.e., LowMC).
- `f`: Whether to do operator fusion to save communication rounds. After operator fusion, we cannot observe the detailed breakdown performance, as the operations are fused together. Default: 0 (i.e., no fusion).

In addition, one can also reuse 2PC subprotocols of FABLE (e.g., deduplication and OPRF evaluation) as building blocks for other protocols. They are implemented under `FABLE/src/GC` and built upon EzPC.

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/.