USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting

Tianyao Gu, *Carnegie Mellon University and Oblivious Labs Inc.;* Yilei Wang, *Alibaba Cloud;* Afonso Tinoco, *Carnegie Mellon University and Oblivious Labs Inc.;* Bingnan Chen and Ke Yi, *HKUST;* Elaine Shi, *Carnegie Mellon University and Oblivious Labs Inc.*

https://www.usenix.org/conference/usenixsecurity25/presentation/gu-tianyao

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: Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting

Tianyao Gu
Carnegie Mellon University
Oblivious Labs Inc.

Yilei Wang
Alibaba Cloud

Afonso Tinoco
Carnegie Mellon University
Oblivious Labs Inc.

Bingnan Chen
HKUST

Ke Yi
HKUST

Elaine Shi
Carnegie Mellon University
Oblivious Labs Inc.

A Artifact Appendix

A.1 Abstract

These artifacts are meant to complement the paper Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting.

Our goal with the artifacts is to provide our opensource implementation of Flexway O-Sort, experiments that allow to easily replicate the results in our paper, and details explanation on how the security goals are achieved by our implementation code. We additionally provide the code of all the baseline / state of the art algorithms used to compare against our implementation.

We provide all our experiment code either as single line commands or as a simple script file to make results simpler to reproduce, and the commands/file should be simple to modify to try new sets of experimental parameters.

Our main goal with the experiments is not to reproduce the exact same numbers as we have in our graphs, as it will vary greatly with hardware used, but to show that the speedup we obtain, as well as the assymptotic behavior is similar to the shown in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. Our code does not have any destructive steps nor does it disable any security mechanisms. We do not export any data and the data used is artificial.

A.2.2 How to access

The code artifacts are available in zenodo: https://zenodo.org/records/14629454

A.2.3 Hardware dependencies

In terms of infrastructure all of our experiments were run on the same machine, with the following specifications:

- Intel Xeon Platinum 8352S processor with 2.2 GHz base frequency
- 1TB DDR4 RAM, 512GB of maximum EPC size.

A.2.4 Software dependencies

Our artifacts are meant to be run under docker, please see the installation steps in order to know how to build and launch the docker container.

A.2.5 Benchmarks

We ran artificial benchmarks on sorting arbitrary arrays comparing with several sorting implementations. And on generating a histogram, initializing an ORAM and load balancing, comparing Flexway O-Sort with bitonic sort.

A.3 Set-up

A.3.1 Installation

First, build the docker image:

Listing 1: build docker images

```
docker build -t cppbuilder:latest
  ./tools/docker/cppbuilder
```

Second, for every test, run docker at the top level of the repo, mounting the ssd for storage and with access to SGX (running with privileged works for this):

Listing 2: start test environment

```
docker run -v /tmp/sortbackend:/ssdmount \
--privileged -it --rm -v $PWD:/builder
  cppbuilder
```

We expect every experiment to be run inside this docker environment.

A.3.2 Basic Test

We include 3 tests here.

(T1) Make sure the code compiles:

Listing 3: compile the code

```
rm -rf build # optional
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
cmake -B build -G Ninja
  -DCMAKE_BUILD_TYPE=Release
ninja -C build
```

(T2) Make sure the unit tests pass:

Listing 4: run unit tests on the code

```
./build/tests/test_algo
```

(T3) Make sure the test enclave compiles and a basic sorting test runs in HW mode:

Listing 5: compile and run a test enclave

```
cd applications/sorting
./algo_runner.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Flexway O-Sort is an assymptotically optimal and concretely efficient sorting algorithm suitable for implementation in hardware enclaves such as Intel SGX having speedups when compared to other state of the art oblivious sorting algorithms for enclaves.
(E1a) and (E1b) show the speedup of Flexway O-Sort against other oblivious sorting algorithms in an enclave setting, supporting the claim (C1).

(C2): Flexway O-Sort is also optimal when the input size is smaller than the EPC size, and still outperforms other state of the art oblivious sorting algorithms for enclaves.
(E4) Shows the speedup of Flexway O-Sort against other oblivious sorting algorithms when the input size is smaller than the enclave size. From it we can see that Flexway O-Sort also outperforms the other state of the art oblivious sorting algorithms, supporting the claim (C2).

(C3): The shuffling algorithm used in Flexway O-Sort outperform the state of the art oblivious shuffling algorithms, such as Waks-in/Waks-out and OrShuffle, regardless of element and input size.
(E2a) and (E2b) show the speedup of Flexway O-Sort against other oblivious shuffling algorithms in an enclave setting, supporting the claim (C3).

(C4): Using Flexway O-Sort as a replacement to commonly used oblivious sorters, such as bitonic sorter, leads to immediate speedups in various applications.
Experiments (E3) show the speedup for ORAM initialization, histogram generation and database join, when we replaced Flexway O-Sort with an external memory optimized recursive bitonic sorter. The results support the claim (C4).

(C5): Our implementation of Sorting are memory oblivious. We don't have any experiment for this, but we encourage looking at both the algorithm description in the paper and the code to conclude that all branches and memory accesses do not depend on private data.

A.4.2 Experiments

Most of our experiments are run across exponentially increasing array sizes from a few thousand to 10^9 elements. We also vary the size of each element, from 8 bytes to 1 kilobyte.

For all the experiments where data < EPC, we run them through ninja test, for all the experiments where EPC < data, we have a file 'applications/sorting/algo_runner.sh' that launches the experiments and needs to be modified accordingly for each experiment, please see the comments in the file for information regarding what the modifications do, and see the file appplications/sorting/README.md for the parameters used for each figure.

(E1a): *[Sorting, EPC < data] [15 human-minutes + 10 compute-hour]: Fig. 7(a)*
How to: Change the parameters in algo_runner.sh to match the ones for Figure 7(a). Run the script. Collect the points and plot them using a log-log scale.
Preparation: Do the initial setup.
Execution: Change the parameters in algo_runner.sh to match the ones for Figure 7(a). Run the script. Collect the points in some file and plot them using a log-log scale.
The script will generate a file name as:

```
<ALGORITHM_NAME>_<MIN_ELEMENT_SIZE>_
<MAX_ELEMENT_SIZE>_<MIN_ARRAY_SIZE>_
<MAX_ARRAY_SIZE>.out.
```

Each line on the file will have the format:

```
<ARRAY_SIZE> <ELEMENT_SIZE> <TIME in
  seconds>
```

Collect these points in some file and plot them using a log-log scale.
Results: After plotting all the graphs, you should obtain a plot similar to Figure 7a. in the paper. You can see that Flexway O-Sort (KWayButterflySort) outperforms the other oblivious sorting algorithms, and has a slope similar to the one of the non-oblivious merge-sort.

(E1b): *[Sorting EPC < data] [15 human-minutes + 10 compute-hour]: Fig. 7(b)*
How to: Repeat the same steps as in E1a, but with the parameters for Figure 7(b).
Preparation: same as E1a.
Execution: same as E1a.
Results: You should obtain a plot similar to Figure 7b. in the paper. You can see that Flexway O-Sort outperforms the other oblivious sorting algorithms.

(E2a): *[Shuffling, EPC < data, varying array size] [15 human-minutes + 10 compute-hour]: Fig. 10(a)*
How to: Repeat the same steps as in E1a, but with the parameters for Figure 10(a).
Preparation: same as E1a.
Execution: same as E1a.
Results: You should obtain a plot similar to Figure 10a. in the paper. You can see that KWayButterfly Shuffle outperforms the other oblivious shuffling algorithms.

(E2b): *[Shuffling, EPC < data, varying item size] [15 human-minutes + 10 compute-hour]: Fig. 10(b)*
How to: Repeat the same steps as in E1a, but with the parameters for Figure 10(b).
Preparation: same as E1a.
Execution: same as E1a.
Results: You should obtain a plot similar to Figure 10b. in the paper. You can see that KWayButterfly Shuffle outperforms the other oblivious shuffling algorithms.

(E3a): *[Application Benchmarks, EPC < data] [15 human-minutes + 3 compute-hour]: Table 3 (128MB EPC)*
How to: Change the parameters in algo_runner.sh to match the ones for Table 3: Benchmark Results for Different Applications. Run the script. Collect the speedup between running the application with Flexway O-Sort and the optimized recursive bitonic sorter in a table similar to Table 3 in our paper.
Preparation: Do the initial setup.
Execution: Change the parameters in algo_runner.sh to match the ones for Table 3: Benchmark Results for Different Applications. Run the script. Collect the points in a table and see that there is always a speedup when using Flexway O-Sort instead of the bitonic sorter.
The script will generate a file in the same format as in E1a. Each benchmark type will have a different format, printing the elements size, some debug information and the time it took to run the algorithm using Flexway O-Sort and the optimized recursive bitonic sorter.
Results: You can see that by replacing the optimized recursive bitonic sorter with Flexway O-Sort there is always a speedup in the different applications tested.

(E3b): *[Application Benchmarks, EPC > data] [15 human-minutes + 3 compute-hour]: Table 3 (EPC > data)*
How to: Build the project as in (T1) and run the binary in ./build/tests/test_apps. The tests will output the runtime for baseline and our as can be seen in table 3.
Preparation: Do the initial setup and compilation in releases mode (T1).
Execution: Build the project as in (T1) and run the binary ./build/tests/test_apps:
Collect the points in a table and see that there is always a speedup when using Flexway O-Sort instead of the optimized recursive bitonic sorter.
Each benchmark type will have a different format, printing the elements size, some debug information and the time it took to run the algorithm using Flexway O-Sort and the optimized recursive bitonic sorter.
Results: You can see that by replacing the optimized recursive bitonic sorter with Flexway O-Sort there is always a speedup in the different applications tested.

(E4): *[Sorting, EPC > data] [15 human-minutes + 3 compute-hour]: Fig. 6(a)*
How to: Run the ninja tests described bellow for the sorting algorithm, described bellow, which uses the parameters for Figure 6(a).
Preparation: Do the initial setup and ninja build.
Execution: Run the following command:

```
./build/tests/test_basic_perf
  --gtest_filter=*TestSortInternal*
```

This will output the time it took to sort arrays of different sizes using several algorithms, when EPC > data. Collect the points in a file and plot them using a log-log scale, similarly to E1a.
Results: You can see the behavior of the different sorting algorithms when EPC > data. You can conclude that Flexway O-Sort (KWayButterflySort) outperforms the other oblivious sorting algorithms, and has a slope similar to the one of the non-oblivious std::sort.

A.5 Notes on Reusability

We included our code as part of the opensource Oblivious Data Structure Library project on https://github.com/odslib/oblsort. We are actively developing it, both in order to do further reserach in oblivious algorithms as well as in order to provide a way for developers to incorporate oblivious algorithms into enclave code. We made our code to be easy integrable into enclave applications. The several application examples we included are a good starting point on how to use this sorting primitive.

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/usenixsec2024/.