USENIX Security '25 Artifact Appendix: OBLIVIATOR: OBLIVIous Parallel Joins and other OperATORs in Shared Memory Environments

Apostolos Mavrogiannakis*
UCSC

Xian Wang*
HKUST

Ioannis Demertzis
UCSC

Dimitrios Papadopoulos
HKUST

Minos Garofalakis
ATHENA Research Center &
Technical University of Crete

A Artifact Appendix

A.1 Abstract

We introduce oblivious parallel operators designed for both non-foreign key and foreign key equi-joins. Obliviousness ensures nothing is revealed about the data besides input/output sizes, even against a strong adversary that can observe memory access patterns. Our solution achieves this by combining trusted hardware with efficient oblivious primitives for compaction and sorting, and two oblivious algorithms: (i) an oblivious aggregation tree, which can be described as a variation of the parallel prefix sum, customized for trusted hardware, and (ii) a novel algorithm for obliviously expanding the elements of a relation. We then implemented our non-foreign key join and foreign key join with the two new algorithms. In the sequential setting, our oblivious join performs 4.6×-5.14× faster than the prior state-of-the-art solution (Krastnikov et al., VLDB 2020) on data sets of size $n = 2^{24}$. In the parallel setting, our algorithm achieves a speedup of up to roughly 16× over the sequential version, when running with 32 threads (becoming up to 80× compared to the sequential algorithm of Krastnikov et al.). Finally, our oblivious operators can be used independently to support other oblivious relational database queries, such as oblivious selection and oblivious group-by, which are implemented and evaluated with complex database queries.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no malicious or destructive operations in the artifact. We propose algorithms and implement systems with a positive impact on preserving data privacy. The artifact does not handle any sensitive data or personal information. All our tests were executed either on synthetic datasets or on publicly available real-world datasets. The ethical implications have also been carefully considered and our work is in full accordance with the 2025 USENIX Security ethics guidelines.

A.2.2 How to access

We open-source all code, datasets and scripts necessary for the evaluation of Obliviator and they can be accessed at [2].

A.2.3 Hardware dependencies

To evaluate the functionality of our artifact, we require a machine with an Intel processor and at least 9GB memory.

A.2.4 Software dependencies

Our implementations are for Linux OS. The artifact requires gcc-7 and g++-7. For all other software dependencies, e.g., the Open Enclave SDK [3], Intel SGX SDK [1], we provide scripts (./scripts/ae-install-dependencies.sh) to install them. In the script, the detailed package versions are chosen for the Ubuntu 20.04 environment.

A.2.5 Benchmarks

We provide ready-to-use datasets for some experiments in our paper. Other experiments need large datasets (hundreds of GB), and we provide scripts to download and process them. The datasets and scripts are in their corresponding sub-folders in ./data denoted by their dataset names. To evaluate the functionality of our artifact, the original hundreds of GB input datasets are not necessary and we provide a test input file (test.txt) within program folders (e.g., ./join/test.txt).

[1] The two first authors contributed equally to this work and their names are listed here alphabetically.

```
1  git clone [GitHub link to the artifact]
2  cd ./obliviator
3  chmod +x ./scripts/ae*
4  ./scripts/ae-install-dependencies.sh
5  source ~/.bashrc
6  ./scripts/ae-intel-sgx.sh
7  cd ~/linux-sgx
8  [Last commands printed in the terminal]
9  cd ~/obliviator
10 ./scripts/ae-basic-test.sh
```

Figure 1: Commands for set-up.

A.3 Set-up

A.3.1 Installation

To install the artifact and dependencies, we provide commands for reference in Figure 1. It may needs 1 hour in total (depending on detailed machines). After the execution of the command at line 6, another suggested command will be printed in the terminal. Please change the directory to another folder as suggested at line 7, and execute the command printed in the terminal. If there is any issue and error regarding the dependency installation, we suggest to execute the commands provided in ./scripts/ae-install-dependencies.sh and ./scripts/ae-intel-sgx.sh line by line to narrow down the detailed incorrect steps.

A.3.2 Basic Test

We provide ./scripts/ae-basic-test.sh to check compilation and see if all software dependencies are installed successfully. Please refer to line 9-10 in Figure 1. Error information will be printed in the terminal if it is not executed successfully.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We propose and implement the new oblivious parallel non-foreign key (NFK) join algorithm. Its function is to join two tables, i.e., matching table rows based on a given join attribute. Its performance with respect to consumed time is expected to be $2.86-77\times$ compared to KKS and KKS* [7], as reported in Section 5.1 of our paper (Table 2, Figure 9 and 10).

(C2): We propose and implement the new oblivious parallel foreign key (FK) join algorithm. Its function is to join a primary key table and a foreign key tables, where the join attribute in the primary table is distinct. Its performance with respect to consumed time is expected to be $1.7-60\times$ compared to Opaque [10], as reported in Section 5.1 of our paper (Table 2 and Figure 11 top left sub-figure).

(C3): We propose and implement the new oblivious filter operator. Its function is to retrieve specific subsets of elements from a database. Its performance with respect to consumed time is expected to be $2.8-52.3\times$ compared to Opaque, as reported in Section 5.2 of our paper (Figure 11 bottom sub-figure).

(C4): We propose and implement the new oblivious aggregation operator. Its function is to generate statistics or a summary of multiple table rows. Its performance with respect to consumed time is expected to be $2.5-53\times$ compared to Opaque, as reported in Section 5.2 of our paper (Figure 11 bottom sub-figure).

(C5): We also implement complex queries to test the above separate operators together. Their function is to obtain the final results of several given operators based on their detailed query requirements [4, 5]. Its performance with respect to consumed time is expected to be $2.58-37.57\times$ compared with Opaque, as reported in Section 5.2 of our paper (Figure 11 top right and bottom sub-figures).

```
1  ./scripts/ae-e1.sh
2  ./scripts/ae.sh
```

Figure 2: Commands for experiments.

A.4.2 Experiments

The evaluation consists of five experiments corresponding to the five claims in Section A.4.1. We provide a script (./scripts/ae.sh) for them, and please refer to the suggested commands in Figure 2 to execute the following five experiments. Alternatively, evaluators can run ./scripts/ae-e1.sh for E1 sub-experiment, ./scripts/ae-e2.sh for E2 sub-experiment, etc., separately.

(E1): [NFK Join] [5 compute-minutes + 9GB memory]: This experiment evaluates the NFK join of Obliviator, and compares it with KKS*. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and KKS*. Our result is expected to be faster than KKS and KKS* and have the suggested speedup as introduced in C1 Section A.4.1. We also conduct E1 on a PC where Obliviator and KKS* take 0.0365 and 0.1294 seconds respectively.

(E2): [FK Join] [5 compute-minutes + 9GB memory]: This experiment evaluates the FK join of Obliviator, and compares it with Opaque. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and Opaque. Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C2 Section A.4.1. We also conduct E2 on a PC where Obliviator and Opaque take 0.0002 and 0.0004 seconds respectively.

(E3): [Filter Query] [5 compute-minutes + 9GB memory]: This experiment evaluates the filter quert of Oblivia-

tor, and compares it with Opaque. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and Opaque. Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C3 Section A.4.1. We also execute E3 on a PC where Obliviator and Opaque take 0.0076 and 0.1288 seconds respectively.

(E4): [Aggregation Query] [5 compute-minutes + 9GB memory]: This experiment evaluates the NFK join of Obliviator, and compares it with Opaque. We are expected to see two floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator and Opaque. Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C4 Section A.4.1. We also execute E4 on a PC where Obliviator and Opaque take 0.1522 and 0.3788 seconds respectively.

(E5): [Complex Query] [5 compute-minutes + 9GB memory]: This experiment evaluates the NFK join of Obliviator, and compares it with KKS and KKS*. We are expected to see six floating point numbers printed in the terminal, which correspond to the consumed time of Obliviator (the first three numbers) and Opaque (the last three numbers). Our result is expected to be faster than Opaque and have the suggested speedup as introduced in C5 Section A.4.1. We also executed E5 on a PC where Obliviator and Opaque spent 1.1705 (the sum of the first 3 numbers) and 3.4669 (the sum of the last 3 numbers) seconds respectively.

A.5 Notes on Reusability.

To reproduce the evaluation of the same experiments with hardware mode in our paper, large machines are required. We are unable to support the original evaluation here due to the large scale and our recent shortage of Intel SGX. The detailed original machine configuration is described in the Section 5 of our paper. Additionally, for reference, the detailed original configuration file for Intel SGX is given in ./script/parallel.conf. If users have an access to such large machines as our paper (i.e., Azure Standard_DC32ds_v3) and want to reproduce, please replace ./[program name]/enclave/parallel.conf with ./scripts/parallel.conf. And, for Obliviator, set the third parameter for oe_create_paralle_enclave in ./[program name]/host/parallel.c to be 0 (replacing the original OE_ENCLAVE_FLAG_SIMULATE), for KKS* (i.e., ./join_kks), set SGX_MODE in Makefile to be HW (or replace Makefile with the content in SGX_Makefile). Note that, to run KKS*, you need to further install the SGX PSW and Openssl, refer [1] or check ./scripts/intel-sgx2.sh .

We improve the scalability of the oblivious operators by designing efficient and fully parallel algorithms, and do not have extra optimizations for its paging overhead. We give our obliviousness proof in our extended paper [8]. And as mentioned in our paper, Intel Pin tool [6] and oblivious constant-time instructions from [9] are also used to help avoid software side-channel attacks for Intel SGX mentioned in our thread model Section 2.

When users test our programs with their own datasets, they are suggested to pay attention to the enclave configuration file parallel.conf and ensure the number of max threads, size of heap, stack (in parallel.conf) and MAX_BUF_SIZE (in parallel.c) are enough.

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/.

References

[1] Intel(r) software guard extensions for linux* os. https://github.com/intel/linux-sgx.

[2] Obliviator. https://zenodo.org/records/15169905.

[3] Open enclave. https://github.com/openenclave/openenclave.

[4] Tpc-h benchmark. http://www.tpc.org/tpch.

[5] AMPLab, University of California, Berkley. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2014.

[6] Intel. Pin - a dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[7] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *VLDB*, 2020.

[8] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. OBLIVIATOR: Oblivious parallel joins and other operators in shared memory environments. Cryptology ePrint Archive, Paper 2025/183, 2025.

[9] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. Distributed & Scalable Oblivious Sorting and Shuffling . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4277–4295, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[10] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 3 2017. USENIX Association.