EnigMap: External-Memory Oblivious Map for Secure Enclaves

Afonso Tinoco, Sixiang Gao, and Elaine Shi, *CMU*

https://www.usenix.org/conference/usenixsecurity23/presentation/tinoco

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.

USENIX'23 Artifact Appendix: EnigMap: External-Memory Oblivious Map for Secure Enclaves

Afonso Tinoco[1,2] Sixiang Gao[1] Elaine Shi[1]

CMU[1], IST[2]

A Artifact Appendix

A.1 Abstract

These artifacts are meant to compliment the paper ENIGMAP: External-Memory Oblivious Map for Secure Enclaves. Our goal with the artifacts is to provide the motivation on why external memory algorithms are relevant for enclaves, our opensource implementation of ENIGMAP, 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 signal and signal-ht code we used to benchmark signal original code. We provide all our experiment code as single line commands to make results simpler to reproduce, and the commands should be simple to modify to try new sets of experimental parameters. We also provide a section explaining at a high level how our implementation works. 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 against signal, as well as the assymptotic behavior is on the order of magnitude shown in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Tag usenix-artifacts in this repo: https://github.com/odslib/EnigMap/tree/usenix-artifacts

Listing 1: download artifacts

```
git clone \
https://github.com/odslib/EnigMap.git \
ARTIFACTS60
cd ARTIFACTS60/
git checkout usenix-artifacts
```

A.2.3 Hardware dependencies

In terms of infrastructure we require 3 types of machines to reproduce our code. Machine A is used mostly for benchmarking SGX, machine B is used to generate a single graph in our experiments, machine C is where most of our experiments were run.

(Machine A) - CPU with SGXv1, configured with at least 128MB of EPC and at least 16GB of RAM. In our experiments we used an intel E2200 processor.

(Machine B) - CPU with SGXv2, configured with 4GB EPC, at least 8GB RAM and an SSD available for storage. This is used mostly for benchmarking ocalls.

(Machine C) - CPU with SGXv2, configured with 192GB EPC, at least 256GB RAM and and SSD available for storage. This is used for most of the experiments. We have a server available with 512GB max EPC and 1TB RAM that can be used to reproduce the experiments for artifact evaluation. Please contact us if access is needed (send us a(n annonymized) public key and we will generate a user in our server)

A.2.4 Software dependencies

Our artifacts are meant to be run under docker, we provide the image use to build and run them under tools/docker/buildDockerImage.sh . Alternatively, we also provide a script to setup a vanila ubuntu 22.04 install (tools/docker/setup_sgx.sh) to run the artifacts.

A.2.5 Benchmarks

We ran baseline benchmarks on private contact discovery using signal and signal-ht (signal-icelake) code. We included them in our repo.

A.3 Set-up

A.3.1 Installation

First, build the docker image:

Listing 2: build docker images

```
#!/bin/bash
```

```
cd ARTIFACTS60/
cd tools/docker/
sh buildDockerImage.sh
```

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 priviledged works for this):

Listing 3: start test environment

```
#!/bin/bash
cd ARTIFACTS60/
docker run -it --rm -m512G \
-v $PWD:/builder -v /mnt/ssd0:/mnt/ssd0 \
--privileged xtrm0/cppbuilder
source /startsgxenv.sh
```

We expect every experiment to be run inside this docker enviroment, additionally some experiments modify configuration files, we assume for every experiment that the configuration files start as they are in the artifact code. We note that some modifications to the config might require deleting the build directory and running cmake again.

A.3.2 Basic Test

We include 3 tests here.

(T1) Make sure the code compiles:

Listing 4: compile the code

```
cmake -B build -G Ninja
ninja -C build
```

(T2) Make sure the tests run:

Listing 5: run unit tests on the code

```
ninja -C build test
```

(T3) Make sure the test enclave compiles and runs in both debug and release mode:

Listing 6: compile and run a test enclave

```
cd applications/benchmark_sgx
make clean
make
./benchmark_sgx.elf
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./benchmark_sgx.elf
```

A.4 Evaluation workflow

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] This section should include all the operational steps and experiments which must be performed to evaluate if your your artifact is functional and to validate your paper's key results and claims. For that purpose, we ask you to use the two following subsections and cross-reference the items therein as explained next.

A.4.1 Major Claims

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Enumerate here the major claims (Cx) made in your paper. Follows an example:

(C1): EnigMap achieves assymptotically speedup on point and batched search over signal's implementation, having speedups at any batch size at a realistic database size of 256 million entries. Our main claim is that switching to EnigMap always provides speedup over signal's original implementation. Additionally, we also claim that for external memory, EnigMap's implementation is assymptotically as described in table 1 in our paper, and concretely efficient.
Experiment (E2) shows the speedup for the database in-RAM case, experiment (E3) shows this claim for the disk eviction case, whose results are ilustrated for SGX1 and SGX2 respectively in figures 3 and 4 in our paper. These two experiments prove the C1 claim.
Further more, we show a detailed view of the costs of external memory and how our optimizations affect them in experiment (E4), reflected in figures 5 and 6 in our paper.
We also show the cost of insertion in experiment, reflected in figure 8 in our paper.

(C2): There is an inherent cost to using EWB that can be reduced with the OCALL approach described. EnigMap's ORAM tree and the OCALL eviction approach provide a way to implicitly store the nonces of evicted pages without any computational overhead compared to EWB, thus saving the space of the evicted nonce table for more EPC pages. We show the inherent costs of EWB and OCALL in experiment (E1), reflected in figure 1 in our paper. We also analyse the effect the of page size for search in experiment (E7) and conclude that roughly 4k is the optimal size for ORAM, reflected in figure 9 in our paper.

(C3): Our faster initialization algorithm is inherently faster than the naive approach of doing sequential insertions. Experiment (E5), reflected in figure 9 in our paper shows these results.

(C4): Our implementation of OBST::Get, OBST::Insert are oblivious. We don't have any experiment for this, but we encorage to look at both the code and the generated code for otree and concluding that all branches inside of otree.hpp::OBST::Get and otree.hpp::OBST::Insert do not depend on private data.

A.4.2 Experiments

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Link explicitly the description of your experiments to the items you have provided in the previous subsection about Major Claims. Please provide your

estimates of human- and compute-time for each of the listed experiments (using the suggested hardware/software configuration above). Follows an example: Most of our experiments are run accross exponentially increasing database sizes from a few bytes to 1TB in size. We include both the time it takes to run the experiments to reach the main conclusions above, as well as the time to fully reproduce the graphs in our paper. We also include the machines required to run each experiment.

(E1.1): *[Benchmark SGX] [20 human minutes + 15 compute-minutes] [Machine A or B or C]* The main goal of this experiment is to show the costs of ocall and and EWB in our paper. We defer computing the cost of EWB to experiment E1.2, altough we analyse it here for clarity.

How to: To run this experiment, compile and run applications/benchmark_sgx . These will show different results based on the processor being used. For SGX1 please only use machine A, for SGX2 use machine B or C. Our graphs in the paper used machine B.

Preparation: Pick a machine, clone the repo and go to the folder applications/benchmark_sgx.

Execution: Compile and run the code:

Listing 7: compile and run a test enclave

```
cd applications/benchmark_sgx
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./benchmark.elf > output.txt
```

Copy paste the values from the benchmark at 4096 bytes to the definition of the data_v2 array at tools/-plot_intrinsics_v2.py. The value for EWB should come from (E1.2). Also update the iops based on the specifications of the SSD being used. Run the script, it should generate a png file with figura 1a.

We generated figure 1b in excel by varying the number of bytes in the encryption and ecall inside of applications/benchmark_sgx/bm.edl.

Results: For the first graph, we expect to see results similar to figure 1a in our paper. For the excel graph, similar to figure 1b.

(E1.2): *[Benchmark SGX] [4 human minutes + 20 compute-minutes] [Machine A or B]* The goal of this experiment is to measure EWB time in a generic SGX machine. It envolves picking an EPC larger than the maximum EPC but smaller than the total RAM, and doing sequential reads and writes to force continuos EWB evictions, and measureing the time based on that (see the function ecall_bm_ewb)

How to: To run this experiment, compile and run applications/benchmark_ewb . These will show different results based on the processor being used. For SGX1 please only use machine A, for SGX2 use machine B. Our graphs in the paper used machine B.

Preparation: Pick a machine, clone the repo and go to the folder applications/benchmark_ewb. Adjust the size of maximum EPC (HeapInitSize and HeapMaxSize) in Enclave.config.xml to be larger than the maximum EPC size on the machine, also adjust the variable ARR_SZ in Enclave/TL/Libcxx.cpp to an appropriate value, so that evictions will occur. The default parameters are all already set for Machine B. We run experiments with different sizes to make it explicit what the page size is.

Execution: Compile and run the code:

Listing 8: compile and run a test enclave

```
cd applications/benchmark_ewb
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./benchmark.elf > output.txt
```

The output should be the total execution time. Divide it by ARR_SZ/4096 to get the cost per page to use in (E1.1).

Results: For the first graph, we expect to see results similar to figure 1a in our paper. For the excel graph, similar to figure 1b.

(E2): *[Point query search inram] [30 human-minutes + 30/120 compute-minutes] [Machine C]:* This experiment computes the cost of point query for a scenario that doesn't need to use disk but needs to do ocalls to in RAM storage. We will explain in here thoroughly how to configure parameters that are further used in other experiments. ODS/common/defs.hpp contains most of the configurable parameters. Take a look at it to see the definitions used and their meanings, configuring experiments is mostly changing this file, as well as changing the Enclave.config.xml used by the experiment. This experiment uses applications/signal.

How to: Go to applications/signal.

Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 0 (search).

Execution: compile and run the enclave:

Listing 9: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_search.py

Results: This will plot graphs similar to figure 3 and 4 in the paper, but not the same, as those graphs use E3.

(E3): *[Point query search with disk] [15 human-minutes + 30/120 compute-minutes] [Machine C]:* This experiment

computes the cost of point query for a scenario that needs to use disk via ocalls.
How to: Go to applications/signal.
Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. set ORAM_USE_INRAM_SERVER to false. Set TEST_SELECTOR to 0 (search).
Execution: compile and run the enclave:

Listing 10: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_search.py
Results: This will plot graphs similar to figure 3 and 4 in the paper. Results should be similar if the same EPC size and maximum RAM were used.

(E4): *[Cost of insertion] [30 human-minutes + 30/120 compute-minutes] [Machine C]:* This shows the cost of insertion queries.
How to: Go to applications/signal.
Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 1 (insertion).
Execution: Compile and run the enclave:

Listing 11: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_insertion.py
The results of this experiment are used to compute the cost in E5 for the naive insertion.
Results: This will plot graphs similar to figure 8 in the paper.

(E5): *[Cost of initialization] [30 human-minutes + 120 compute-minutes/ 24 compute-hours] [Machine C]:* This shows the cost of initialization queries.
How to: Go to applications/signal.
Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 2 (initialization).
Execution: Compile and run the enclave:

Listing 12: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_initialization.py
Use the results of E4 to get the costs for naive initialization (cost of 1 insertion * number of elements in the database)
Results: This will plot graphs similar to figure 7 in the paper. We can see that the fast initialization has a speedup over the naive initialization.

(E6): *[Optimization breakdown] [45 human-minutes + 20 compute-minutes] [Machine B]:* This experiment analyses an execution trace of the code to generate figures 5 and 6.
How to: This uses ods outside of enclave to generate flame graphs of the execution profile.
Preparation: Edit buildtype.cmake to change the build type to debug.
Execution: Compile and run the proiling tests:

Listing 13: compile and run a test enclave

```
rm -rf build
cmake -B build -G Ninja
ninja -C build
./build/tests/improv*_none
mv ./q*/f*/flame*-chrome.json \
./q*/f*/improv*_none.json
./build/tests/improv*_packing
mv ./q*/f*/flame*-chrome.json \
  ./q*/f*/improv*_packing.json
./build/tests/improv*_filecache
mv ./q*/f*/flame*-chrome.json \
  ./q*/f*/improv*_filecache.json
./build/tests/improv*_bucketcache
mv ./q*/f*/flame*-chrome.json \
  ./q*/f*/improv*_bucketcache.json
./build/tests/profiling_test
```

Open the flamegraphs in chrome and analyse the total time in each phase. To plot the graphs, one can use plot_relative_performance.py or plot_relative_performance_large.py.
With the default config, we get the plot in figure 5. Run the experiment again with a larger database size (edit the respective .cpp files for the tests) to get the case for figure 6.

Results: This will plot graphs similar to figure 5 and 6 in the paper. We can see the effect of different optimizations.

(E7): *[Optimal page size] [30 human-minutes + 30 compute-minutes] [Machine B or C]:* This experiment is meant to analyse what is the optimal page size used for insertion. We used machine B in our experiments to enforce disk swap at smaller sizes, but this can also be ran on machine C.

How to: Go to applications/signal.

Preparation: To run this experiment, configure Enclave.config.xml to match the EPC size of the machine used, configure ORAM_SERVER__DIRECTLY_CACHED_LEVELS to the maximum value that fits within EPC. Additionally set ORAM_USE_INRAM_SERVER to true. Additionally set TEST_SELECTOR to 0 (search). For each experiment, configure ORAM_SERVER__LEVELS_PER_PACK to different values (1 corresponds to 296B page, 2 to 824B, 3 to 1880B, 4 to 3993B and 5 to 8216 in our plot).

Execution: Compile and run the enclave:

Listing 14: compile and run a test enclave

```
cd applications/signal
make clean
SGX_MODE=HW SGX_PRERELEASE=1 make
./signal.elf > output.txt
```

You can plot the results manually or use tools/scripts/-plot_search_pagesize.py

Results: This will plot graphs similar to figure 9 in the paper, we can see that the optimal page size is 4kb.

In all of the above blocks, please provide indications about the expected outcome for each of the steps (given the suggested hardware/software configuration above).

A.5 Notes on Reusability

We provide our code as an opensource Oblivious Data Structure Library project on https://github.com/odslib/EnigMap. 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 example inside the folder applications/signal is meant to be easily integrable as a binary search tree or map into any SGX project.

We hope that our artifacts can be used by industry to provide fast private contact discovery in messaging applications such as signal. We also hope that the code in EnigMap can be further improved and serve as baseline for further research in oblivious algorithms, all the oblivious primitives we developed in EnigMap can easily be integrated into other enclave or oblivious algorithms projects.

A.6 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/.