USENIX Security '24 Artifact Appendix: PerfOMR: Oblivious Message Retrieval with Reduced Communication and Computation

Zeyu Liu
Yale University

Eran Tromer
Boston University

Yunhao Wang
Yale University

A Artifact Appendix

A.1 Abstract

Our artifact is a C++ library implementing the constructions PerfOMR1, PerfOMR2 in [1]. Our main claims produced from this artifact are the detector runtime of these constructions in Table 1 and Table 2.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact should not incur any risk to the evaluators regard of the machines security, data privacy or ethical concerns. The payload we use to simulate the public data published on the bulletin board is drawn from random distributions. Note that we are using an old version of OpenSSL, but it does not require the root privilege and should only be built in the directory specified by the evaluator. Despite all these, we still recommend the evaluators open a **fresh** GCP instance without testing our code using their own machines.

A.2.2 How to access

The main code is public on github under ObliviousMesageRetrievel repo.

A.2.3 Hardware dependencies

Our main benchmarks should be able to be reproduced on a normal Google Compute Cloud e2-standard-4 instance type (4 hyperthreads of an Intel Xeon 3.10 GHz CPU with 16GB RAM),

A.2.4 Software dependencies

On a Google Compute Cloud e2-standard-4 instance, we run the benchmarks with boot disk configured with Ubuntu 20.04 LTS operating system and a 256GB disk. Notice that the disk memory is used to store the large database our experiments run against.

We also rely on the following softwares and libraries:

- C++ build environment
- CMake build infrastructure
- SEAL library 4.1 and all its dependencies Notice that we made some manual change on SEAL interfaces to facilitate our implementation and thus a built-in dependency of SEAL is directly included under 'build' directory.
- PALISADE library release v1.11.2 and all its dependencies, as v1.11.2 is not publicly available anymore when this repository is made public, we use v1.11.3 in the instructions instead.
- NTL library 11.4.3 and all its dependencies
- OpenSSL library on branch OpenSSL_1_1_1-stable We use an old version of OpenSSL library for plain AES function without the complex EVP abstraction.

A detailed installation script is provided in the README.md file in our artifact. the datasets we use are simulated directly when running the experiements and thus no third-party models/datasets are used.

A.2.5 Benchmarks

We benchmark the main scheme PerfOMR1 (Section 5.3) and the alternative scheme PerfOMR2 (Section 6). The parameters we use are: the number of transactions in the dataset $N = 2^{19}, 2^{21}, 2^{23}$, the number of pertinent messages for the recipient $k = \bar{k} = 50, 100, 150$, and the batch size $v = 8$.

A.3 Set-up

A.3.1 Installation

```
# If permission required, please add sudo
# before the commands as needed

sudo apt-get update && sudo apt-get install
build-essential # if needed
sudo apt-get install autoconf # if no autoconf
sudo apt-get install cmake # if no cmake
sudo apt-get install libgmp3-dev # if no gmp
sudo apt-get install libntl-dev=11.4.3-1build1
# if no ntl
sudo apt-get install unzip # if no unzip
```

	Detector Runtime (ms/msg)	Clue Key Size (kB)	Clue Size (Bytes)	Detector Key Size (MB)	Digest Size (Bytes/msg)	Recipient Runtime(ms)
PerfOMR1 [1, Sec 5]	**7.31**	2.13	2181	171	2.57	37
PerfOMR2 [1, Sec 6]	39.64	**0.56**	**583**	140	1.03	20

Table 1: Comparison of cost metrics. Costs are per recipient. The bulletin contains $N = 2^{19}$ messages, of which $\bar{k} = k = 50$ are pertinent to the recipient. ms/msg and Bytes/msg are all amortized over N messages. Each message has 612 bytes of payload.

		$k = \bar{k} = 50$			
	N	Amortized runtime (ms/msg)	Total runtime (s)	Amortized digest size (Bytes/msg)	Total digest size (MB)
PerfOMR1	2^{19}	7.31	3931.65	2.57	1.35
	2^{21}		15868.37	0.48	
	2^{23}		64701.09	0.12	
PerfOMR2	2^{19}	39.64	20953.45	1.03	0.54
	2^{21}		82826.57	0.26	
	2^{23}		330985.56	0.06	

		$N = 2^{19}$			
	$k = \bar{k}$	Amortized runtime (ms/msg)	Total runtime (s)	Amortized digest size (Bytes/msg)	Total digest size (MB)
PerfOMR1	50	7.31	3931.65	2.57	1.35
	100	9.29	4874.03	4.71	2.47
	150	11.15	5847.55	9.34	4.67
PerfOMR2	50	39.96	20953.45	1.03	0.54
	100	41.58	21797.76	1.63	0.81
	150	42.85	22465.38	2.16	1.08

Table 2: Performance of our constructions when N and $k = \bar{k}$ varies.

```
# If you have the PERFOMR_code.zip directly,
# put it under ~/OMR and unzip it into
# ObliviousMessageRetrieval dir, otherwise:
mkdir -p ~/OMR
cd ~/OMR
wget https://github.com/ObliviousMessageRetrieval/
ObliviousMessageRetrieval/raw/3c4245d66806f032517
a9f20447ca78d5419d380/PERFOMR_code.zip

unzip PERFOMR_code.zip

# change build_path to where you want the
# dependency libraries installed
OMRDIR=~/OMR
BUILDDIR=$OMRDIR/ObliviousMessageRetrieval/build

cd $OMRDIR && git clone -b v1.11.3
https://gitlab.com/palisade/palisade-release
cd palisade-release
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=$BUILDDIR
make
make install

# Old OpenSSL used for plain AES function
# without EVP abstraction
cd $OMRDIR && git clone -b OpenSSL_1_1_1w
https://github.com/openssl/openssl
cd openssl
./config --prefix=$BUILDDIR
make
make install

# Optional
# Notice that although we 'enable' hexl via
# command line, it does not take much real
# effect on GCP instances and thus does not
# have much impact on our runtime
cd $OMRDIR && git clone --branch v1.2.3
https://github.com/intel/hexl
cd hexl
cmake -S . -B build -DCMAKE_INSTALL_PREFIX=$BUILDDIR
cmake --build build
cmake --install build

cd $OMRDIR/ObliviousMessageRetrieval/build
mkdir ../data
mkdir ../data/payloads
mkdir ../data/clues
mkdir ../data/cluePoly
mkdir ../data/processedCM
cmake .. -DCMAKE_PREFIX_PATH=$BUILDDIR
make
```

A.3.2 Basic Test

The instruction to run a test on our construction is as follows:

```
./OMRdemos <perfomr1/perfomr2>
<number_of_cores> <number_of_messages_in_bundle>
<number_of_bundles> <number_of_pert_msgs>
```

For example, a valid sanity test could be:

```
cd $BUILDDIR
./OMRdemos perfomr1 1 2 32768 50
```

The expected output should look like this:

```
Preparing database and paramaters...
Pertient message indices: [ 3558 ... 32215 ]
```

```
/
| Encryption parameters :
|   scheme: BFV
|   poly_modulus_degree: 32768
|   coeff_modulus size: ... bits
|   plain_modulus: 65537
\
Database and parameters prepared.

Preprocess switching key time: 264900272 us.
ClueToPackedPV time: 160386979 us.
PVUnpack time: 675661388 us.
ExpandedPVToDigest time: 218222953 us.

Detector running time: 965717410 us.
Result is correct!
```

A.4 Evaluation workflow

A.4.1 Major Claims

Our benchmark claims are all in Table 1 and Table 2. The major one to be reproduced is the detector runtime (note that clue key and clue sizes can both be calculated with the parameters we wrote in our paper; digest size can be as well (with a modulus switching of the final BFV ciphertexts to ~30 bit ciphertext modulus); the recipient runtime is not a major claim of our paper and it is not optimized for), up to some testing variation. In particular, the "detector runtime" column in Table 1 and the "amortized runtime" in Table 2.

A.4.2 Experiments

Before executing any experimental scripts in this section, we assume that one has finished the installation steps in Appendix A.3.1. All the experiments are run with 16GB RAM and 256GB disk. The expected outcomes should be similar to the one given under Appendix A.3.2

Notice that the runtime of our experiments are quite long (the main scheme takes most of the time, while the preparation of the dataset also takes one to dozens of hours, depending on how long the dataset is), **we highly recommend one to use** screen **command** to detach all running scripts from the current session and put it on the back-end, so that one is still able to re-attach it after the current session times out (which does happen a lot when running our experiments). We also recommend one to initialize several fresh instances and run those experiments in parallel.

(E1): Runtime scaling with the number of transactions N: the following run scripts aim to reproduce the detector runtime stated in the top half of Table 1.

Execution:

```
./OMRdemos perfomr1 1 8 65536 50  # around
3 CPU hours
./OMRdemos perfomr1 1 8 262144 50 # around
6 CPU hours
./OMRdemos perfomr1 1 8 1048576 50 # around
21 CPU hours
./OMRdemos perfomr2 1 8 65536 50 # around 8
CPU hours
./OMRdemos perfomr2 1 8 262144 50 # around
26 CPU hours
./OMRdemos perfomr2 1 8 1048576 50 # around
100 CPU hours
```

Results: After seeing the detector running time (in us) in the log, divide it by the number of transactions (notice that the total number of transactions equals to number_of_messages_in_bundle multiplied with number_of_bundles). For example, if by running script `./OMRdemos perfomr1 1 8 65536 50` with log: `Detector running time: 3831655159 us`, the amortized runtime should be $3831655159/(65536*8) = 7308\text{us} = 7.31\text{ms}$.

(E2): Runtime scaling with the number of pertinent messages $k(\bar{k})$: the following run scripts aim to reproduce the detector runtime stated in the bottom half of Table 1.

Execution:

```
./OMRdemos perfomr1 1 8 65536 50 # around 3
CPU hours
./OMRdemos perfomr1 1 8 65536 100 # around
3.5 CPU hours
./OMRdemos perfomr1 1 8 65536 150 # around
4 CPU hours
./OMRdemos perfomr2 1 8 65536 50 # around
6.5 CPU hours
./OMRdemos perfomr2 1 8 65536 100 # around
7 CPU hours
./OMRdemos perfomr2 1 8 65536 150 # around
7.5 CPU hours
```

Results: The calculation steps are exactly the same as above.

A.5 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/.

References

[1] Z. Liu, E. Tromer, and Y. Wang. Perfomr: Oblivious message retrieval with reduced communication and computation. Cryptology ePrint Archive, Paper 2024/204, 2024. Full version of this paper. Available on print: https://eprint.iacr.org/2024/204.