GigaDORAM: Breaking the Billion Address Barrier

Brett Falk, *University of Pennsylvania;* Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang, *University of California, Los Angeles*

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

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: GigaDORAM: Breaking the Billion Address Barrier

Brett Falk*
University of Pennsylvania

Rafail Ostrovsky*
UCLA

Matan Shtepel*
UCLA

Jacob Zhang*
UCLA

A Artifact Appendix

A.1 Abstract

The artifact for the USENIX'23 paper "GigaDORAM: Breaking the Billion Address Barrier" is this GitHub repository. The repository contains a C++ implementation of the GigaDORAM protocol, benchmarking scripts, and explanations helpful to reproducing the experiments in the paper. The explanations are composed of a standard README file and a **detailed video walkthrough** which we believe to be the most convenient way to install GigaDORAM and reproduce our results.

A.1.1 What is GigaDORAM?

GigaDORAM is a 3-party state-of-the-art Distributed ORAM protocol (DORAM) protocol. DORAM is a stateful multiparty cryptographic protocol. We envision the protocol holding a secret shared state, `Memory`, with `N` 0-initialized cells, `Memory[0],...,Memory[N-1]`. An execution of the protocol takes secret shared variables X_{query}, Y_{new}, `IsWrite` as input. The output of the protocol is secret shared `Memory[`X_{query}`]`. If `IsWrite=1`, a stateful update `Memory[`X_{query}`]` := Y_{new} is preformed. Under certain non-collusion assumptions, an execution of the protocol does not reveal *any* information to the participating parties.

GigaDORAM is a 3-party DORAM protocol specialized for the low-latency, large `N` setting. In these settings, GigaDORAM significantly outperforms previous protocols. In other settings, GigaDORAM preforms comparably to previous protocols. See the paper, and in particular Section 9, for more details.

A.2 Description & Requirements

Roughly speaking, we benchmarked GigaDORAM in 2 different settings

- *Single machine tests:* we execute GigaDORAM through 3 processes on the same machine. This enables us to artifically restrict the network between the machines processes via the `tc` command and benchmark the performance of GigaDORAM in a variety network settings.
- *Multi machine tests:* we execute GigaDORAM on 3 different AWS EC2 instances in the same AWS region. These tests are meant to demonstrate the "real world" potential of GigaDORAM.

In Section A.3 and Section A.4 we show how to set-up and execute both kinds of tests, respectively. Again, we suggest that the best way to follow along with setup, installation, and experiment-replication is our **detailed video walkthrough**

A.2.1 Security, privacy, and ethical concerns

`single_server_experiments.py` runs `sudo tc qdisc replace dev lo root` to simulate network latency and bandwidth limits on the loopback interface, which can slow down other programs running on the machine. We recommend running on a dedicated VM. If anything strange happens, the network changes can be undone with `sudo tc qdisc del dev lo root`

A.2.2 How to access

Our artifact can be accessed via this GitHub repository. It is not our development repository and will not be evolving.

A.2.3 Hardware dependencies

For evaluation, a processor supporting the Intel SSE2 instruction set is and we recommend at least 8 CPU cores and 8GB of RAM. While local evaluation of our artifact is possible, benchmarking on AWS is necessary for replicating our results. If AWS credits are not available to reviewers, please reach out to us via the anonymous submission portal.

A.2.4 Software dependencies

- A Linux machine with processor supporting the Intel SSE2 instruction set.
- `sudo` access is unfortunately required; this isn't an issue on AWS.

[*] Authors are in alphabetical order.

- For compilation, the EMP-toolkit library. Follow the instructions at EMP-toolkit repository to install EMP and its dependencies.
  - EMP's dependencies are extremely basic (`python3 cmake git build-essential libssl-dev`) and are all needed to build GigaDORAM. Those can be installed using the `apt` package manager.
- On a typical Linux system there are no dependencies needed to run the compiled binary.

A.2.5 Benchmarks

No data is needed to run our tests.

A.3 Set-up

In this section we describe set-up for single-server GigaDORAM tests, multi-server GigaDORAM tests, and tips for optionally testing other DORAM constructions.

A.3.1 Installation

Single server tests.

1. Clone our repository
2. Run `compile.sh`. If this fails, make sure you've installed EMP-toolkit.

Multi server tests. The multi-machine tests are designed to be run on AWS EC2. *Warning*: Follow the directions below carefully; it is easy to miss something which will cause the benchmarks to not be able to run.

- You will need choose an AWS region and create and start AWS EC2 instances named `DORAM_benchmark_1`, `DORAM_benchmark_2`, `DORAM_benchmark_3` in that region.
  - Use the same SSH key for access to all 3 instances, as it will be an argument to the script later.
  - Set security group settings to allow TCP traffic between the 3 instances.
  - We used `c5n.metal` instances, which guaranteed that our parties were not running on the same physical host, and also provided high multi-threaded performance.
  - We used Ubuntu 22.04 but any modern enough Linux distribution should work. Recall that Intel processors are required.
- For most tests, the instances should be created in a cluster placement group. A quick how to is covered in our video tutorial. For more information about cluster placement groups, see here.
- In addition to the requirements for single server tests, the AWS CLI (package `awscli` in apt) needs to be installed on the machine used for builds.
- After installing, configure your region and access keys with `aws configure`.
- Additionally, you will need to add these lines to `~/.ssh/config/` on the build machine: `Host * StrictHostKeyChecking no`

  Without disabling StrictHostKeyChecking, the experiment script will be unable to ssh to new hosts in the background, since user input would be required to continue connecting to a new host.
- Nothing needs to be installed on the benchmark instances!

Optional: other DORAM constructions. In this section, we briefly comment on the procedure we took to install and set-up other DORAM constructions.

- *DuORAM*: We benchmark DuORAM via their well documented dockerization.Detailed information can be found in their README.
- *3PC-ORAM*: We benchmarked 3PC-ORAM via the convenient dockerization graciously provided by the DuORAM team. Again, details can be found in the README.
- *Sqrt ORAM, Circuit ORAM, fss-FLORAM, cprg-FLORAM*: We benchmark Sqrt ORAM, Circuit ORAM, fss-FLORAM, and cprg-FLORAM via Doerner and shelats' original code. Due to a reliance on the somewhat aged obliv-c framework, we ran into some difficulties running their code. We try to give some helpful tips here (note: some of this steps may be redundant or incomplete – we simply note here what worked for us):
  - We started two Ubuntu 18.04.6 EC2 instances in a cluster placement group, one to be the "server" and the other to be the "client"
  - To install the needed old version of `ocaml`, run `sudo apt install opam`, `opam switch create 4.06.0`, and `eval $(opam env --switch=4.06.0)`
  - To install the needed old version `gcc`, `sudo apt install -y gcc-9 g++-9 cpp-9`
  - Then follow the obliv-c README to install.
  - Then follow the FLORAM README to install
  - *PFEDORAM*: PFEDORAM is proprietary, and we obtained benchmarks directly from Bingsheng Zhang, one of the authors of the paper.

A.3.2 Basic Test

Single server.

```
./benchmark_doram_locally.sh 100us 10Gbit -prf-circuit-filename LowMC_reuse_wires.txt -build-bottom-level-at-startup false -num-query-tests 10 -log-address-space 20 -num-levels 3 -log-amp-factor 4 -num-threads 4
```

Multi server.

```
./run_3_server_experiment.sh ../my_key.pem -prf-circuit-filename LowMC_reuse_wires.txt -build-bottom-level-at-startup false -num-query-tests 10 -log-address-space 20 -num-levels 3 -log-amp-factor 4 -num-threads 20
```

Optional: other DORAM constructions. Other DORAMs contain simple tests in their respective READMEs

A.4 Evaluation workflow

A.4.1 Major Claims

The main claim of our paper is the performance of GigaDORAM that can be seen in various network settings / configurations. These can be found in Figure 5, Figure 6, Table 1, and Table 2.

A.4.2 Experiments

Below we provide descriptions of how to execute our main results. In the repositories README we also provide descriptions on how to run any set of parameters in both the single server and multiserver setting

(E1): *Reproducing Figures 6a and 6b, 10 human-minutes + 1.5 compute hour + 20 compute threads and 20 GB RAM. These tests reproduce the performance of GigaDORAM in simulated varying latency and varying bandwidth settings.* :

Preparation: noted in Section A.3.

Execution: The single server experiments are run by `./single_server_experiments.py` which prints usage information with the `-h` flag.

To reproduce the tests used to generate the GigaDORAM data in Figures 6a and 6b in the paper, run

```
./single_server_experiments.py Figure6a
./single_server_experiments.py Figure6b
```

We ran these tests on a machine with 96 CPUs and saw best results with 20 threads per party (which is the default). If you are running on a smaller machine, you should use a num_threads which is less than 1/3 the number of CPUs available, for example:

```
$ ./single_server_experiments.py Figure6a
    --threads 2
$ ./single_server_experiments.py Figure6b
    --threads 2
```

Results: The concatenated output from all experiments will be written to `single_server_results/doram_timing_report${i}.txt` for party i in 1, 2, 3.

These files are are human readable and formatted in blocks as, for example:

```
DORAM Parameters
Number of queries: 1000
Build bottom level at startup: 0
Log address space size: 16
Data block size (bits): 64
Log linear level size: 8
Log amp factor: 4
Num levels: 3
PRF circuit file: LowMC_reuse_wires.txt
Num threads: 1

Timing Breakdown
Total time including builds: 2.89467e+06 us
Time spent in queries: 2.83478e+06 us
Time spent in query PRF eval: 1.21668e+06 us
Time spent querying linear level: 751191 us
Time spent in build PRF eval: 8775 us
Time spent in batcher sorting: 0 us
Time spent building bottom level: 0 us
Time spent building other levels: 20811 us

SUMMARY
Total time including builds: 2.89467e+06 us
Total number of bytes sent: 2.7828e+07
Queries/sec: 345.462
```

The `SUMMARY` section is the most important to look at, as it gives the total time and communication to run the test.

WARNING: single_server_experiments.py runs `sudo tc qdisc replace dev lo root` to simulate network latency and bandwidth limits on the loopback interface, which can slow down other programs running on the machine. We recommend running on a dedicated VM. If anything strange happens, the network changes can be undone with `sudo tc qdisc del dev lo root`

(E2): *Multi server tests, reproducing Figure 5, Table 1, and Table 2, 10 human-minutes + 1 compute-hour + 3 strong, running AWS EC2 machines*

Preparation: Noted in Section A.3.

Execution: `./multi_server_experiments.py -h` for syntax help. Run the following experiments one by one, checking the results output after each

```
$ ./multi_server_experiments.py Figure5
    ../my_pem_file.pem
$ ./multi_server_experiments.py Table1
    ../my_pem_file.pem
$ ./multi_server_experiments.py Table2
    ../my_pem_file.pem
$ ./multi_server_experiments.py Figure8
    ../my_pem_file.pem
```

Results: The concatenated output from all experiments will be written to `multi_server_results/doram_timing _report${i}.txt` for party i in 1, 2, 3, following the same format as the single server results.

(*Optional* E3): *reproducing the results of other DORAMs [1.5 human-hours (including setup) + 4 compute-hour + several AWS EC2 instances (number pending on how many tests are ran in parallel)]: benchmark the performance (in queries/sec) of other DORAM constructions.*

Preparation: Noted in Section A.3.

Execution: In this section, we briefly comment on the procedure we took to benchmark other DORAM constructions. We describe below which command For the settings we tested each construction in and additional discussions, please see Section 9, Figures 5 and 6, and Appendix E of the paper. As each figure describes the benchmark setting, here we only describe the code commands we used.

- *DuORAM*: For varying `numops` and `size`, we summed `./run_experiment read size numops preproc 3P` and `./run_experiment readwrite size numops online 3P` to account for both the online and offline costs of protocol.
- *3PC-ORAM*:Using `./run-experiment size numops` we benchmarked 3PC-ORAM's *reads*, which are no more expensive than writes.
- *Sqrt ORAM, Circuit ORAM, fss-FLORAM, cprg-FLORAM*: On the "server" EC2 machine call `./bench_oram_write -e ADDRESS_SPACE_SIZE -o TYPE -i 1024` and then on the "client" EC2 machine call `./bench_oram_write -e ADDRESS_SPACE_SIZE -o TYPE -i 1024 -c ADDRESS_OF_SERVER` where `ADDRESS_SPACE_SIZE` is `N`, not `log_N`, `TYPE` can be either `{sqrt, circuit, fssl, fssl_cprg}`, `-i` marks the number of writes to be done, and `ADDRESS_OF_SERVER` is the IP address of server (make sure AWS security group is set to allow for TCP traffic).
- PFEDORAM: PFEDORAM is proprietary, and we obtained benchmarks directly from Bingsheng Zhang, one of the authors of the paper.

Results: Each of these constructions respective READMEs explains the output format.

A.5 Notes on Reusability

We believe the GigaDORAM implementation and benchmarks we provide reflect the performance of GigaDORAM accurately in various network settings. However, our implementation is *not* production ready. For instance, it may contain timing attacks and lacks several important optimizations.

It is possible to execute GigaDORAM on data other than the (dummy) benchmark data, but at this stage that might require some slight understanding of our codebase.

To start, we suggest https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/doram/doram_array.h in our repository which, via several subclasses, implements the GigaDORAM protocol. For black-box usage, other than the constructor, the only function from this class that that should be called publicly is `read_and_write`. Unfortunately, because GigaDORAM is a multi-machine, multi-threaded program, running GigaDORAM is not as simple as calling the constructor and then the method. For an example we suggest https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/test/doram/doram.cpp which shows initialization the resources (e.g. Psuedorandom function seeds) necessary for executing GigaDORAM, proceeds to construct a GigaDORAM object, and then calls `read_and_write` on it repeatedly. To see how `doram.cpp` gets called, we suggest https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/run_3_server_experiment.sh which, for example, is called by https://github.com/jacob14916/GigaDORAM-USENIX23-Artifact/blob/main/multi_server_experiments.py.

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/.