USENIX Security ’25 Artifact Appendix: Practical Keyword Private Information Retrieval from Key-to-Index Mappings

Meng Hao[1], Weiran Liu[2], Liqiang Peng[2], Cong Zhang[3] (✉), Pengfei Wu[1], Lei Zhang[2], Hongwei Li[4], and Robert H. Deng[1]

[1]School of Computing & Information Systems, Singapore Management University
[2]Alibaba Group
[3]Institute for Advanced Study, BNRist, Tsinghua University
[4]Peng Cheng Laboratory

A Artifact Appendix

A.1 Abstract

In this artifact appendix, we introduce the minimal hardware and software requirements for the artifact. Since the artifact related to our paper is written in pure Java programming language, the implementation is cross-platform and evaluators can try the artifact on any platform(s) with JDK 17 (or later) installed. We also introduce the source code structure and the ways to run unit tests. We describe how to reproduce the performance reports shown in our paper. Large-scale experiments require relatively large RAM, but evaluators can still try small-scale experiments even on laptops. We welcome suggestions and performance reports on other platforms with future reproducibility.

A.2 Description & Requirements

The artifact contain the implementations of our three constructions: KPIR$^{\mathsf{kvs}}$, KPIR$^{\mathsf{hash}}$, and KPIR$^{\mathsf{index}}$, together with the baseline construction ChalametPIR (CCS’24). Evaluators can use the artifact to reproduce Table 1 to Table 6 shown in our paper. We note that evaluators cannot reproduce column “**Memory**” shown in Table 2. The reason is that the tool for reporting the memory usage is jol[1], whose license is not compatible with our license. If evaluators are familiar with Java, it would be relatively easy to manually include related dependencies, add required source codes, and obtain the memory usage report based on the artifact.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact of our paper has been open-sourced and merged into mpc4j (version 1.1.3)[2]. The stable Zenodo URL for this artifcate is available at `https://zenodo.org/records/14722434`. Evaluators can either visit the stable Zenodo URL or the stable version under tag v1.1.3 to reproduce the experiment results shown in the paper.

Package “edu.alibaba.mpc4j.s2pc.pir.cppir.ks.simple” of the submodule mpc4j-s2pc-pir contains the implementations of our three constructions: KPIR$^{\mathsf{kvs}}$ (source codes start with SimpleNaive), KPIR$^{\mathsf{hash}}$ (source codes start with SimpleBin), KPIR$^{\mathsf{index}}$ (source codes start with SimplePgm). Package “edu.alibaba.mpc4j.s2pc.pir.cppir.ks.chamelet” of the same submodule contains the implementation of the baseline construction ChalametPIR (CCS’24). Evaluators can also find unit tests for all client-preprocessing keyword PIR protocols in the test folder of package “edu.alibaba.mpc4j.s2pc.pir.cppir.ks”. Evaluators can find an example configuration file in the resource folder, which is used for running and reproducing performance reports.

A.2.3 Hardware dependencies

Since implementations related to our paper is written in pure Java programming language, the artifact is expected to support any platforms with JDK 17 (or later) installed. We have tested functionalities of the artifact on 64-bit macOS, Ubuntu, and CentOS systems.

We run our experiments on two machines, each equipped with an Intel Core i9-9900K processor running at 3.6GHz and 128GB of memory. Evaluators may encounter java.lang.OutOfMemoryError when running artifacts on small RAM machines under large parameters.

✉Corresponding author
[1]`https://openjdk.org/projects/code-tools/jol/`

[2]`https://github.com/alibaba-edu/mpc4j/`

A.2.4 Software dependencies

The only software dependency is JDK 17 (more later). We note that unit tests also cover other keyword PIR protocol functionalities in the artifact. Therefore, evaluators may encounter errors when running unit tests for protocols that are not related to our paper. Nevertheless, evaluators can still try other protocol implementations after following the guidelines described in the readme to install and configure required native libraries.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The artifact is implemented in a way that chooses the most efficient protocol parameters. Unfortunately, this means that reproducing some performance reports shown in our paper requires modifying source codes. Therefore, we highly recommend loading the artifact using IntelliJ IDEA, the leading Java IDE, so that one can later modify source codes more easily. Evaluators can follow Section **Development** shown in the readme to configure, compile, and run unit tests in IntelliJ IDEA. Evaluators who are familiar with Java can alternatively download and compile the artifact with the following steps.

1. Clone the repository: git clone https://github.com/alibaba-edu/mpc4j.git.
2. Go to the root path: cd mpc4j.
3. Package the code to get the jar file: mvn package.

A.3.2 Basic Test

In package "edu.alibaba.mpc4j.s2pc.pir.cppir.ks", evaluators can find two unit tests, simple/SimpleCpKsPirParamsTest.java, CpKsPirTest.java, in the test folder. Directly pressing the green arrows showing on the left of these source codes can run these unit tests.

Unit tests in simple/SimpleCpKsPirParamsTest.java are used to reproduce columns "**# Row**" and "**# Column**" shown in Table 3, Table 4, and Table 5, while the column "Exp. Rate" can be manually computed based on the report. Since we implement the matrix operations in the transposed form (due to efficiency consideration), we reversely reported "**# Row**" and "**# Column**" in our paper.

Unit tests in CpKsPirTest.java are used to test the functionality of the artifact. Directly run them in IntelliJ IDEA and see if all tests for KPIR$^{\text{kvs}}$ (SIMPLE_NATIVE), KPIR$^{\text{hash}}$ (SIMPLE_BIN), KPIR$^{\text{index}}$ (PGM_INDEX) can pass.

A.4 Evaluation workflow

A.4.1 Major Claims

To reproduce the performance results shown in our paper, evaluators need to generate the jar file (with the name mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar) by executing mvn package, and run it with two progresses on a single machine, or two progresses on two machines connected by the network. Evaluators also need to create a config file with suitable parameters. Below is a template configuration file.

```
# server information
server_name = server
server_ip = 192.168.1.1
server_port = 9002

# client information
client_name = client
client_ip = 192.168.1.2
client_port = 9003

# append string in the output file
append_string = example

# protocol type
pto_type = SINGLE_CP_KS_PIR

# protocol config
entry_bit_length = 256
server_log_set_size = 22,22,20,18
query_num = 100
parallel = false

# protocol name
single_cp_ks_pir_pto_name = PGM_INDEX
# or CHALAMET, SIMPLE_NAIVE, SIMPLE_BIN
```

(C1): Unit tests in simple/SimpleCpKsPirParamsTest.java can reproduce Table 3, Table 4, and Table 5.

(C2): Running jars with suitable parameters can reproduce Table 1.

(C3): Running jars with suitable source code modifications can reproduce Table 2 and Table 6 (except column "**Memory**", see Section A.2).

A.4.2 Experiments

(E1): [10 human-minutes + 0.1 compute-hours]: Reproduce Table 3, Table 4 and Table 5.
How to: See Section A.3.2.
Results: The numbers shown in the console should be the same as ones listed in Table 3, Table 4, and Table 5.

(E2): [15 human-minutes + 3 compute-hour]: Reproduce Table 1.

How to: Run two processes on two machines connected via network cards, forming a realistic LAN network with a bandwidth of 2.5Gbps and an RTT latency of 0.4ms.

Preparation: Setup two machines. Create a config file shown in Section A.4.1 with correct server/client information, required entry bit length, set sizes, query nums, and protocol names. We assume the config file name is YOUR_CONFIG_FILE_NAME.conf. Place the jar file and the config file under the same path.

Execution: Open one terminal on each of the machines, one for the server and one for the client. Switch to the dictionary where mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar and YOUR_CONFIG_FILE_NAME.conf are located. For the server's terminal, execute java -jar mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar config_file_name.conf server. For the client's terminal, execute java -jar mpc4j-s2pc-pir-1.1.3-jar-with-dependencies.jar YOUR_CONFIG_FILE_NAME.conf client. Wait until the processes finish. The performance reports are located in the folder "temp".

Results: The performance reports should be similar with ones listed in Table 1.

(E3): [30 human-minutes + 6 compute-hours]: Reproduce Table 2 and Table 6.

How to: Same as E3.

Preparation: To reproduce Table 2, evaluators need to manually assign the value of ε range to 4, 8, 16, 32 sequentially by changing line 30 of the code simple/SimplePgmCpKsPirDesc.java with suitable ε range and re-compile the jar file. To reproduce Table 6, evaluators need to manually assign the dimension of LWE as 1408 by changing line 44 of the code simple/SimpleNaiveCpKsPirConfig.java, simple/SimpleBinCpKsPirConfig.java, and SimplePgmCpKsPirConfig.java from `this(GaussianLweParam.N_1024_SIGMA_6_4);` to `this(GaussianLweParam.N_1408_SIGMA_6_4);` and re-compile the jar file. The remains are same as E3.

Execution: Same as E3.

Results: The performance reports should be similar with ones listed in Table 2 and Table 6.

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/usenixsec2025/`.