USENIX'23 Artifact Appendix: Linear Private Set Union from Multi-Query Reverse Private Membership Test

Cong Zhang[1,2], Yu Chen[3,4,5] (✉), Weiran Liu[6], Min Zhang[3,4,5] and Dongdai Lin[1,2]

[1]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China*
{zhangcong,ddlin}@iie.ac.cn
[3]*School of Cyber Science and Technology, Shandong University, Qingdao 266237, China*
[4]*State Key Laboratory of Cryptology, P.O. Box 5159, Beijing 100878, China*
[5]*Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao 266237, China*
yuchen.prc@gmail.com, zm_min@mail.sdu.edu.cn
[6]*Alibaba Group*
weiran.lwr@alibaba-inc.com

A Artifact Appendix

A.1 Abstract

We introduce our open-source project mpc4j, an efficient and easy-to-use Secure Multi-Party Computation (MPC) library mainly written in Java. Package psu in mpc4j-s2pc-pso of mpc4j contains the implementations, along with configurations needed to replicate our experiments from Section 6. In particular, our artifact supports running and comparing Private Set Union (PSU) protocols with element set sizes up to 2^{20} on machines having 128GB memory. We also provide guidelines for installing dependencies and compiling native libraries needed by mpc4j on different platforms, including x86_64 MacBook, MacBook with M1 chip, Ubuntu 20.04, and CentOS 8. The project is licensed under Apache License 2.0. The source code is available online at https://github.com/alibaba-edu/mpc4j. The stable version for the artifact evaluation is available at https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.4.

In this artifact appendix, we first introduce the minimal hardware and software requirements to get performance reports shown in our paper using mpc4j. Then, we introduce how to install and run mpc4j on different platforms. We note that there are some performance gaps between different platforms, and having complete comparisons for different protocols is very challenging. Aside from that, mpc4j still tries to provide a library for having relatively unified comparisons. We welcome suggestions and performance reports on other platforms with future reproducibility.

A.2 Description & Requirements

We introduce our open-source project mpc4j (Multi-Party Computation for Java), an efficient and easy-to-use Secure Multi-Party Computation (MPC) library mainly written in Java. mpc4j aims to provide an academic library for researchers to study and develop MPC and related protocols in a unified manner. As mpc4j tries to provide state-of-the-art MPC implementations, researchers could leverage the library to have quick and unified comparisons between the proposed and existing protocols.

Package psu in mpc4j-s2pc-pso of mpc4j contains the implementations, along with configurations needed to replicate our experiments from Section 6. Existing Private Set Union (PSU) implementations are under different MPC frameworks and different experimental settings. After carefully studying existing open-source codes, we fully re-implement exisiting PSU protocols and their underlying basic protocols using Java. Evaluators can test PSU protocols on mpc4j by simply using different configuration files. All experiment results shown in Section 6 of our paper are obtained by running mpc4j.

Evaluators can compile and run mpc4j on different 64-bit platforms. We provide guidelines for installing dependencies and compiling native libraries needed by mpc4j on different platforms, including x86_64 MacBook, MacBook with M1 chip, Ubuntu 20.04, and CentOS 8. Note that successfully running all PSU experiments with large element size (i.e., $n = 2^{20}$) requires 128GB RAM. We run our experiments on a single Intel Core i9-9900K with 3.6GHz and 128GB RAM. We note that there are some performance gaps between different platforms. We welcome suggestions and performance reports on other platforms with future reproducibility.

In the full version of our paper, we further provide experiment results on two PSU applications, namely IP blacklist aggregation and Private ID. The related source code has been merged into version v1.0.5[1].

[1] https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.5

A.2.1 How to access

mpc4j is available online on GitHub at https://github.com/alibaba-edu/mpc4j. Evaluators can visit the stable version v1.0.4 (https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.4) to reproduce the experiment results shown in the paper.

A.2.2 Hardware dependencies

mpc4j currently support 64-bit macOS, Ubuntu, and CentOS systems. Evaluators may meet errors when compiling mpc4j on a 32-bit or less system. The reason is that mpc4j uses some 64-bit Single instruction, multiple data (SIMD) operations.

A.2.3 Software dependencies

mpc4j leverages native C/C++ codes to speed up cryptographic operations. The native codes and Java codes are interacted by the Java Native Interface (JNI) technique.

We separate native C/C++ codes into two modules, namely mpc4j-native-tool and mpc4j-native-fhe. mpc4j-native-tool contains native codes for basic cryptographic operations, while mpc4j-native-fhe contains native codes for Fully Homomorphic Encryption (FHE) using SEAL[2]. All basic cryptographic operations in mpc4j-native-tool have alternative pure-Java implementations in mpc4j with the same functionalities and the same data representations. Note that if evaluators only run mpc4j for PSU, there is no need to install SEAL and compile mpc4j-native-fhe. mpc4j-native-tool relies on the following C/C++ libraries:

- GMP (https://gmplib.org/): An efficient library for operations with arbitrary precision integers, rationals, and floating-point numbers.
- NTL (https://libntl.org/): A high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers and for vectors, matrices, and polynomials over the integers and over finite fields, developed by Victor Shoup (https://shoup.net/). Note that one can further introduce GF2X (https://gitlab.inria.fr/gf2x/gf2x) for more efficient operations in a Galois Field. However, since the installation procedure for GF2X is rather complicated, we use NTL by default.
- MCL (https://github.com/herumi/mcl): A portable and fast pairing-based cryptography library. MCL also includes fast Elliptic Curve implementations, especially the optimized implementation for the elliptic curve secp256k1.
- libsodium (https://doc.libsodium.org): A modern, easy-to-use software library for encryption, decryption, signatures, password hashing, and more. libsodium includes efficient implementations for the elliptic curve Curve25519 with APIs for X25519 and Ed25519.
- OpenSSL (https://www.openssl.org/): a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication. OpenSSL includes many efficient cryptographic primitive implementations.

[2] https://github.com/microsoft/SEAL

A.3 Set-up

A.3.1 Installation

Installing mpc4j-native-tool might be a bit complicated for ones who are not that familiar with Unix-like systems, since the procedures differ across platforms. The documentation (README.md) in package mpc4j-native-tool provides instructions for installing mpc4j-native-tool on macOS (x86_64 / aarch64), Ubuntu, and CentOS, respectively.

A.3.2 Basic Test

We develop mpc4j using Intellij IDEA (https://www.jetbrains.com/idea/) and CLion (https://www.jetbrains.com/clion/). After successfully compiling mpc4j-native-tool (Please see readme.md in these modules for more details on how to compile them), evaluators only need the community version of Intellij IDEA to run all basic tests.

Evaluators need to configure IDEA with the following procedures so that IDEA can link to the complied mpc4j-native-tool native libraries.

1. Open "Run->Edit Configurations..."
2. Open "Edit Configuration templates..."
3. Select "JUnit".
4. Add the following command into "VM Options": -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH.

After that, evaluators can run tests of any submodule by pressing the green arrows showing on the left of the source code in test packages. See Section **Demonstration** of readme.md in mpc4j on details for running the tests.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): In our paper, we claimed that we fully re-implement state-of-the-art PSU protocols and their underlying basic protocols using Java. This can be verified by running basic tests in psu (See Section A.3.2 for details), or running experiments with different configuration files (See Section A.4.2 for details).

(C2): In our paper, we claimed that although there is some performance gap, most basic operations in Java and C/C++ have similar performances. This can be verified by running all efficient tests in mpc4j-common-tool (test classes with names end with "EfficiencyTest"). For example, try running "PrpEfficiencyTest" in the package edu.alibaba.mpc4j.common.tool.crypto.prp of the submodule mpc4j-common-tool, evaluators can see the performance comparisons between using AES provided by Java and by AES-NI invoked with JNI.

A.4.2 Experiments

(E1): *[Generate jar file] [5 human-minutes + 5 compute-minutes]: Generate mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar containing the main function entry.*
How to: On the charms bar of IDEA, evaluators can find a button with name "Maven". Press that botton, double-click "mpc4j -> Lifecycle -> package", IDEA would automatically compile and generate mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar containing the main function entry.
Preparation: Evaluators need to successfully running basic tests before generating the jar file.
Execution: Just double-click "mpc4j -> Lifecycle -> package".
Results: The generated file would be located in "mpc4j/mpc4j-s2pc-pso/target".

(E2): *[(optimal) Config network settings] [5 human-minutes + 1 compute-minute]: Config network settings using tc.*
How to: Open a terminal, and execute the following command: "tc qdisc add dev lo root netem rate 10Mbit latency 80ms". Then, the local network is configured as 10Mbit bandwidth with 80ms latency. Evaluators can try other network settings with other parameters, e.g., 100Mbit/80ms, 1Gbit/40ms, 10Gbit/0.02ms.
Preparation: None
Results: Execute "sudo tc qdisc show dev lo" to see if the network is configured correctly.

(E3): *[Run experiments] [10 human-minutes + 5 compute-hour]: Run experiments using different configuration files.*
How to: Open two terminals, one for the PSU server and one for the PSU client. Switch to the dictionary where mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar located (Evaluators can also copy the generated jar file to other dictionaries). For the server's terminal, execute "java -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 -jar mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar CONFIG_SERVER_FILE.txt". For the client's terminal, execute "java -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 -jar mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar CONFIG_CLIENT_FILE.txt". The corresponding server/client configuration files are in "mpc4j-s2pc-pso/conf/psu". Note that evaluators must first run server and then run client.
Preparation: None.
Note: It would take a long time to run if the network has limited bandwidth, long latency, and/or a large set size. See the performance results of our paper to estimate the total running time. Evaluators may find that the setup of SKE-PSU time is quite different from the result presented in Table 3 of our paper. This is because in the paper, we assume Boolean multiplication triples are pre-computed offline and stored locally in a temporary file. Therefore, the setup phase only contains loading Boolean multiplication triples into the memory. In our artifact, we dynamically generate Boolean multiplication triples in the setup phase using silent Oblivious Transfer techniques. In the full version of the paper, we provide the triple generation costs for SKE-PSU, which would be similar to the costs in the setup phase evaluators obtained using the artifact.
Results: Java would run the experiments and generate the performance reports under the current dictionary.

A.5 Notes on Reusability

Evaluators can check and modify server/client configuration files to change IP addresses, port numbers, the element byte length used for PSU. We also provide other configuration examples (marked with #) for specific PSU protocols.

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/.