

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Enhanced Private Set Union from Secret-shared Private Membership Test

Meng Hao, *Singapore Management University;* Guodong Wang, *University of Electronic Science and Technology of China;* Xinpeng Yang, *Nanyang Technological University;* Pengzhi Xing and Hanxiao Chen, *University of Electronic Science and Technology of China;* Tianwei Zhang, *Nanyang Technological University;* Haiyang Xue and Guomin Yang, *Singapore Management University;* Hongwei Li, *University of Electronic Science and Technology of China;* Robert H. Deng, *Singapore Management University*

https://www.usenix.org/conference/usenixsecurity26/presentation/hao

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology



USENIX Security '26 Artifact Appendix: Enhanced Private Set Union from Secret-shared Private Membership Test

Meng Hao[1], Guodong Wang[2], Xinpeng Yang[3], Pengzhi Xing[2], Hanxiao Chen[2,✉], Tianwei Zhang[3], Haiyang Xue[1], Guomin Yang[1], Hongwei Li[2,✉], Robert H. Deng[1]
[1]Singapore Management University
[2]University of Electronic Science and Technology of China
[3]Nanyang Technological University

A Artifact Appendix

A.1 Abstract

In this artifact appendix, we provide a self-contained roadmap for evaluating the artifact associated with our paper. We describe the required hardware, software, and network-emulation configuration, and explain how evaluators can build and run the artifact either natively on a Linux machine or inside the provided Docker environment. All implementations are written in C++ and include scripts for building dependencies, running basic functionality tests, and reproducing the main timing and communication results reported in the paper. Full-scale experiments may require substantial CPU time, memory, and privileged Linux network namespace support, the goal of this appendix is to make the artifact straightforward to inspect, build, execute, and compare with the paper results.

A.2 Description & Requirements

The artifact contains the implementations of our two ePSU variants: ePSU_fast and ePSU_low, together with the implementations of their main building blocks, including ssPMT, psPMT, and RPMT. Evaluators can use the artifact to reproduce the end-to-end ePSU results shown in Figure 24, Table 2, and Table 3 of our paper. These results report the running time and communication cost of the full ePSU protocol under different input sizes and network settings. Evaluators can also use the artifact to reproduce the component-level results shown in Figure 21 and Figure 22.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact of our paper is publicly available on Zenodo at https://zenodo.org/records/20411092, with the version-specific DOI 10.5281/zenodo.20411092. This Zenodo record provides a stable archived version of the artifact associated with the camera-ready paper. Evaluators can download the artifact package from this record and use it to reproduce the experimental results reported in our paper.

The artifact root directory contains the README, Dockerfile, and two main implementation directories: ePSU_fast and ePSU_low. The ePSU_fast directory contains the fast ePSU implementation and the benchmark for ePSU. The ePSU_low directory contains the low-communication ePSU implementation and the related benchmark. Evaluators can follow the README to build the artifact, run the basic functionality tests, and reproduce the results for Figure 21, Figure 22, Figure 24, Table 2, and Table 3.

A.2.3 Hardware dependencies

The implementations are written in C++ and run on standard x86-64 Linux machines. The experimental results reported in the paper were obtained on a server equipped with two Intel Xeon Platinum 8368Q processors and 1 TB of RAM. For fair comparison, all baseline protocols were re-run under the same experimental setup, and both our protocols and the baselines were executed using a single thread.

Evaluators do not need access to this exact server to check the functionality of the artifact. However, full reproduction of the absolute running time reported in the paper may depend on using a machine with comparable CPU and memory resources. On different machines, evaluators should mainly compare the communication cost and the overall performance trends.

A.2.4 Software dependencies

The artifact is intended to run on Linux. We provide a Dockerfile based on Ubuntu 24.04, which installs the required system packages, including CMake, a C++ compiler, Git, Python 3, OpenMP, OpenSSL, GMP, NASM, iproute2, and other build tools. We recommend evaluators use the Docker environment to avoid manually configuring these dependencies.

The artifact also relies on third-party cryptographic libraries, including volepsi and secure-join. These libraries are downloaded and built automatically by the provided setup scripts. The benchmark scripts that emulate LAN and WAN settings require Linux network namespaces and the tc command from iproute2. Therefore, full benchmark reproduction should be run with sudo privileges or inside a privileged Docker container. No proprietary software, external dataset, or machine-learning model is required.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We recommend that evaluators build the artifact using the provided Dockerfile, since it installs the required system packages and compiles both ePSU implementations in a clean Linux environment. The artifact can also be built natively on Linux by running the setup scripts in the two implementation directories.

From the artifact root directory, evaluators can build and start the Docker container as follows:

1. Build the Docker image: `docker build -t epsu-sspmt .`
2. Start the container: `docker run -rm -it -privileged epsu-sspmt bash`

The `-privileged` option is required for the full benchmark scripts, which emulate LAN and WAN network settings using Linux network namespaces and `tc netem`.

Evaluators who prefer native installation can build the two implementations separately as follows:

1. Install the required system packages: `sudo apt-get update && sudo apt-get install -y -no-install-recommends ca-certificates openssl build-essential cmake git libtool autoconf automake pkg-config iproute2 python3 sudo nasm libssl-dev libgmp-dev wget libfmt-dev && sudo update-ca-certificates`
2. Build the fast implementation: `cd ePSU_fast && bash setup.sh`
3. Build the low-communication implementation: `cd ../ePSU_low && bash setup.sh`

A.3.2 Basic Test

After building the artifact, evaluators can run a small two-party ePSU execution to check that the required software components are built and functioning correctly.

For the fast implementation, run:

1. `cd ePSU_fast`
2. `bash ePSU_bench.sh 14`

For the low-communication implementation, run:

1. `cd ../ePSU_low`
2. `bash ePSU_bench.sh 14`

A successful run should end with:
`[SUCCESS] Finished.`
The output should also contain timing and communication information for the executed protocol. These numbers are only used to check functionality and are not expected to match the full paper results exactly.

A.4 Evaluation workflow

A.4.1 Major Claims

To reproduce the performance results shown in our paper, evaluators need to build the artifact and run the benchmark scripts provided in the two implementation directories, ePSU_fast and ePSU_low. The scripts execute two protocol parties on a single machine and record running time and communication cost under different input sizes and emulated network settings.

(C1): The end-to-end performance of our two ePSU variants can be reproduced by running the ePSU benchmark scripts in ePSU_fast and ePSU_low. The reproduced results correspond to Figure 24, Table 2, and Table 3 in the paper.

(C2): The performance of the PMT building blocks used in our ePSU constructions can be reproduced by running the ssPMT benchmark in ePSU_fast and the psPMT benchmark in ePSU_low. The reproduced results correspond to Figure 21 in the paper.

(C3): The performance of the RPMT building block can be reproduced by the artifact. This claim is evaluated by running the RPMT benchmark scripts in both ePSU_fast and ePSU_low. The reproduced results correspond to Figure 22 in the paper.

A.4.2 Experiments

(E1): End-to-end ePSU performance [20 human-minutes + several compute-hours]: Reproduce the end-to-end performance results of `ePSU_fast` and `ePSU_low` which corresponds to Figure 24, Table 2, and Table 3 in the paper.

Preparation: Use a Linux machine or a privileged Docker container, since the benchmark scripts use Linux network namespaces and `tc netem` to emulate LAN and WAN settings. Build both `ePSU_fast` and `ePSU_low` before running the experiment.
Execution: Run the following commands:

1. `cd ePSU_fast`
2. `sudo bash ePSU_batch_bench.sh`
3. `cd ../ePSU_low`
4. `sudo bash ePSU_batch_bench.sh`

Results: The scripts generate timestamped log directories such as `logs_ePSU_YYYYMMDD_HHMMSS`. The reproduced numbers should be compared with Figure 24, Table 2, and Table 3.

(E2): ssPMT/psPMT component performance [15 human-minutes + several compute-hours]: Reproduce the performance results of the PMT building blocks which corresponds to Figure 21 in the paper.
Preparation: Build both implementations. The fast implementation evaluates `ssPMT`, while the low-communication implementation evaluates `psPMT`.
Execution: Run the following commands:

1. `cd ePSU_fast`
2. `sudo bash ssPMT_batch_bench.sh`
3. `cd ../ePSU_low`
4. `sudo bash psPMT_batch_bench.sh`

Results: The scripts generate log directories report the running time and communication cost of the PMT components under different input sizes and network settings. The results should be similar to the trends reported in Figure 21.

(E3): RPMT component performance [15 human-minutes + several compute-hours]: Reproduce the performance results of the RPMT building block which corresponds to Figure 22 in the paper.
Preparation: Build both implementations.
Execution: Run the following commands:

1. `cd ePSU_fast`
2. `sudo bash RPMT_batch_bench.sh`
3. `cd ../ePSU_low`
4. `sudo bash RPMT_batch_bench.sh`

Results: The scripts generate logs report running time and communication cost for RPMT. The reproduced results should be compared with Figure 22. Absolute running time may vary across machines, while communication cost and scalability trends should remain consistent.

A.5 Notes on Reusability

The artifact is organized into two independent implementations, `ePSU_fast` and `ePSU_low`, so future users can study, modify, or benchmark each variant separately. The benchmark scripts can also be adapted to evaluate different input sizes or network settings by changing the set-size list and the `tc netem` parameters in the scripts.

The component benchmarks for `ssPMT`, `psPMT`, and `RPMT` may be useful beyond reproducing the paper results. Researchers can use these components to compare alternative building blocks or to evaluate new protocol optimizations under the same LAN and WAN emulation framework. The artifact does not require external datasets, so new experiments can be created by adjusting the protocol parameters and rerunning the provided scripts.

A.6 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/usenixsec2026/.