USENIX Security '26 Artifact Appendix: Rarus: A Succinct and Efficient Range Proof for Polynomial-based Vector Commitment

Xinyang Yang, Wenjie Qu, Yanpei Guo, Jiaheng Zhang

A Artifact Appendix

A.1 Abstract

The artifact provides the complete C++ implementation of Rarus, a zero-knowledge polynomial range proof protocol, along with baseline comparisons (Bulletproofs and Plookup). It leverages the high-performance `mcl` library for underlying MSM and pairing functions. The artifact includes all necessary cryptographic gadgets, such as non-recursive Bivariate NTT, polynomial operations, and KZG commitments. It is designed to reproduce the core benchmarking claims made in the paper regarding Prover time, Verifier time, and Proof size.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The stable and permanently archived version of this artifact is available on Zenodo at https://doi.org/10.5281/zenodo.20322978. The active development repository, which can evolve during the evaluation discussion phase, is hosted on GitHub at https://github.com/Montayang/rangeproof.

A.2.3 Hardware dependencies

None. The artifact runs entirely on standard CPU-based environments. It does not require GPUs, Trusted Execution Environments, or any custom hardware.

A.2.4 Software dependencies

The system is tested on standard vanilla Linux distributions (e.g., Ubuntu 22.04 LTS or 24.04 LTS). It requires standard C++ build tools and cryptographic libraries, which can be installed via the following commands:

```
sudo apt-get update
sudo apt-get install -y libgmp-dev libssl-dev
sudo apt-get install -y cmake make g++ git python3
```

The protocol strictly depends on the `mcl` library (V3+), which will be compiled from source during the set-up phase.

A.2.5 Benchmarks

The experimental evaluation relies on the baseline implementations of Bulletproofs and Plookup included in this artifact (`bulletproof.cpp`, `plookup.cpp`). No external datasets or pre-trained models are required. The benchmark executable automatically generates random polynomial inputs based on the parameters (b,k,ℓ) specified in Section 6 of our paper to measure the protocol's efficiency.

A.3 Set-up

A.3.1 Installation

1. Clone the Rarus repository and enter the directory.
2. Clone the `mcl` library directly into the project root:
   `git clone https://github.com/herumi/mcl.git`
3. Compile the `mcl` core library:
   `cd mcl && make clean && make -j4 && cd ..`
4. Build the main project:
   ```
   mkdir build && cd build
   cmake .. && make -j4 && cd ..
   ```

A.3.2 Basic Test

Navigate to the `build` directory and execute the main binary: `./main`. The expected successful output will display a sequence of completion logs (e.g., `Setup Completed.`, `Step1 Completed.`), concluding with the Prover Time, Verifier Time, and Proof Size without any linking or segmentation faults.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Rarus achieves specific performance metrics (Prover Time, Verifier Time, and Proof Size) as reported in Section 6 of the paper. This is proven by the core experiment (E1).

A.4.2 Experiments

(E1): Automated Benchmarks [5 human-minutes + 30 compute-minutes + 500MB disk]: Evaluate the core protocol and baselines to reproduce the paper's results.
How to: Use the provided Python master script to automatically inject parameters, recompile, and execute the benchmarks.
Preparation: Complete all steps in the Set-up and Installation section to initialize the `build` directory and compile the `mcl` core library.
Execution: From the project root directory, run the master script: `python3 run_experiments.py`. The script handles incremental compilation and execution automatically.
Results: The terminal will output formatted ASCII tables that directly mirror the execution time (Prover/Verifier Time in seconds) and proof size (in KB) results presented in Table 8, Table 9, and the data points in Figures 3 and 4 of our paper.
Important Note on Evaluation Variance: As discussed during the AE process, evaluators should expect variations in absolute execution times. Because the underlying cryptographic operations (e.g., MSMs and NTTs) are heavily CPU-bound, absolute wall-clock times will naturally differ across various hardware architectures (e.g., differences in microarchitecture or clock speed). Additionally, standard Linux environments may exhibit run-to-run variance (typically 5%–15%) due to OS scheduling and thermal dynamics. However, the core claims of our paper—the *relative asymptotic scaling ratios* and the mathematically deterministic *Proof Sizes* (e.g., 1.53125 KB)—will remain strictly consistent across all environments.

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/usenixsec2026/.