USENIX Security '24 Artifact Appendix: <Notus: Dynamic Proofs of Liabilities from Zero-knowledge RSA Accumulators>

Jiajun Xin Arman Haghighi Xiangan Tian Dimitrios Papadopoulos

The Hong Kong University of Science and Technology

A Artifact Appendix

A.1 Abstract

In this artifact evaluation, we evaluate the implementation of the Notus prototype and its underlying RSA accumulators, SNARK circuits, smart contracts to verify the proofs and low-level optimizations for fast multi-exponentiations. Most parts of our experiment are developed using Golang, and the smart contract is based on Solidity. The evaluation results are expected to show the practicability of the Notus system and our optimizations for both Notus and multi-exponentiations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

These experiments do not contain any destructive steps nor harm the server running the code. We want to stress that the RSA accumulator parameters we use are generated by ourselves and shall not be used in the production environment.

A.2.2 How to access

The main part of the experiment is in this project (https://github.com/notus-project/rsa_accumulator/tree/v0.1.0), including the prototype of Notus, and its implementation using zero-knowledge RSA accumulators, specific SNARK design, the code for proving and verification, and the smart contract to verify the proofs.

A.2.3 Hardware dependencies

Our main program can be run on any server with commodity hardware and 20GB disk free space for the very basic procedure. Because our code is highly optimized for parallelization and uses a precomputation table to optimize performance further, to verify all the claims of our results, we require 40 GB disk free space, 32 threads, and 128GB memory.

A.2.4 Software dependencies

The main part of the experiment is based on Golang 1.19 or above. It can be run either on Windows, Linux or MacOS as long as it supports Golang. To verify the gas cost of our Ethereum smart contract, we provide the code based on Foundry, which requires Linux or MacOS (also possible on Windows but requires additional setup processes). We also claim the same gas estimation result can be achieved using any smart contract test bed, for example, the web-based testbed Remix (https://remix.ethereum.org/). For the simplicity of the evaluator, a sample data that passes the smart contract verification is provided in https://github.com/notus-project/solidity_contract/blob/v0.1.0/test/Verifier.t.sol.

A.2.5 Benchmarks

There are many ways to test the gas cost of a smart contract, and the results might jitter slightly due to the state of the blockchain, the level of optimization, and different versions of the platform. For simplicity of verifying the gas cost of this project's smart contract (https://github.com/notus-project/solidity_contract/blob/v0.1.0/src/Verifier.sol), we provide the code to verify the gas cost of our smart contract in https://github.com/notus-project/solidity_contract/tree/v0.1.0.

A.3 Set-up

To run the main part of the experiment, Golang with version 1.19 or above is required. Detailed instructions can be found from https://go.dev/dl/.

To use Foundry to estimate the gas cost of our smart contract, Foundry is required. Detailed instructions can be found from https://book.getfoundry.sh/getting-started/installation.

A.3.1 Installation

To run the main experiment, clone the GitHub project, use #go mod tidy to install the necessary Golang packages, and

use #go build to build executable files.

To run the gas estimation of our smart contract, clone the GitHub project and checkout to the correct branch.

A.3.2 Basic Test

To run the main experiment, in folder "rsa_accumulator", run the compiled file "rsa_accumulator" and input "1" to run a basic test.

Because our multi-exponentiation optimizations can also be used for generous purposes, we listed and benchmarked it independently in https://github.com/notus-project/multiexp/tree/v0.1.0. To run the benchmark of the multi-exponentiation optimizations, input #go test -bench . to run all the benchmarks in the folder directly using Golang's official benchmark framework.

To run the gas estimation of our smart contract, in folder "solidity_contract", run the following to get the result: #forge test

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The multi-exponent optimizations achieve better efficiency than the basic exponentiation algorithm based on Montgomery modular multiplications. The expected results are depicted in Section 6, Table 2.*

(C2): *RSA accumulator performance. This benchmark tests the performance of RSA accumulators both in a single thread and multiple threads and the time to retrieve a precomputed membership proof. The expected results are depicted in Section 6, Figure 8.*

(C3): *Notus system performance. This benchmark tests the performance of the Notus system, including a component profiler in a single thread, the number of constraints of its SNARK circuit as well as proving time under different numbers of users. As a comparison, a simple SNARKed-Merkle tree is also benchmarked. The expected results are depicted in Section 6, Figure 8, Figure 9, and Table 3.*

(C4): *Gas cost of our smart contract. This benchmark tests the gas cost of verifying the smart contract, which verifies Groth16 proof and PoKE proof for RSA accumulators, respectively. The expected results are depicted in Section 6: the Gas cost per audit proof is only* 750*K (*270*K for verifying a Groth16 proof and* 480*K for verifying two PoKEs).*

A.4.2 Experiments

(E1): *[The multi-exponent optimizations benchmarks] [1 human-minutes + 5 compute-minute]: requires at least 2 threads to see the benefit of parallelization. The experiments test from 1 thread to 16 threads.*
Preparation: N/A
Execution: *In the folder of multiexp, execute #go test -bench .*
Results: *We observe that with our optimizations DoubleExponent is around* 30% *faster and FourfoldExponent around* 60% *faster than their original counterparts. Additionally, we report the performance of FourfoldExponent when combined with a precomputation table that includes a precomputation of every single bit; we refer to this as the precomputeFourfoldExponent function.*

(E2): *[RSA accumulator benchmarks.] [1 human-minutes + 2.5 compute-hour]: requires 32 threads to see the benefit of parallelization and 120GB memory to load the precomputation tables.*
Preparation: N/A
Execution: *In the folder of rsa_accumulator, execute #./rsa_accumulator and execute condition 2 and 4.*
Results: *These two experiments illustrate the performance of our RSA accumulators.*

(E3): *[Notus benchmarks.] [1 human-minutes + 6 compute-hour]: requires a single thread for "Single Core Component Profiler" and 32 threads to benchmark the performance of the Notus system with parallelization, 120GB memory to load the precomputation tables, and 40 GB disk space to store the SNARK circuit.*
Preparation: N/A
Execution: *In the folder of rsa_accumulator, execute #./rsa_accumulator and execute condition 3 (Single Core Component Profiler) and 5 (Notus under different group size in parallel).*
Results: *Condition 3 illustrates the percentage of running time for membership precomputation in Notus system with a single thread, and condition 5 shows the benchmark of the Notus system*

(E4): *[Smart contract benchmarks.] [5 human-minutes + 1 compute-minute]: requires single thread.*
Preparation: N/A
Execution: *In the folder of solidity_contract, execute #forge test.*
Results: *It outputs the estimated gas cost of the smart contract.*

A.5 Notes on Reusability

As we mentioned previously, our multi-exponent optimizations are designed for general-purpose computations. Its code can be further extended for more threads and "denser" precomputation tables.

The (zero-knowledge) RSA accumulator, together with its highly parallelized membership-proof precomputation scheme, can be used independently as an efficient implementation of RSA accumulators. Besides, the smart contract to verify the Groth16 proofs is generated automatically using the gnark library, which means we can generate smart contracts

for other circuits beyond Notus easily.

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/usenixsec2024/.