USENIX Security '26 Artifact Appendix: Concretely Efficient Blind Signatures Based on VOLE-in-the-Head Proofs and the MAYO Trapdoor

Carsten Baum[1], Marvin Beckmann[1], Ward Beullens[2], Shibam Mukherjee[3,4], Christian Rechberger[3,5]

Technical University of Denmark[1], IBM, Zürich[2], Graz University of Technology[3], Know Center, Graz[4], TACEO, Graz[5]

A Artifact Appendix

A.1 Abstract

This artifact appendix contains implementations of our proposed Blind signatures in the main paper. With this artifact, we support our performance claims and provide a functional open access implementation of our work. Easing compilation effort, we also provide a `bench_all_bs.sh` script generating a table corresponding to Table 3 in our main work. A `Docker` file has also been provided to alleviate the complications of package dependencies. The implementation uses optimized C/C++ codebase and combines into a Rust framework. The MAYO implementation is from the most recent NIST submission[1]. The framework for the VOLEitH proofs is from the most recent NIST submission of FAEST[2]. Our ZK-proof is implemented in C++ by modifying the codebase of FAEST. The C/C++ components are made accessible in Rust using foreign function interfaces (FFIs) using the bindgen[3] crate. The high-level aspects of our Blind signature are in Rust. The Rust build system (`cargo`) compiles and wraps the underlying C/C++ code. The code can be run directly through the Rust environment. The protocol logic in Rust can be modified without recompiling the underlying C/C++ code. Any changes to the underlying C/C++ code require a clean build of the FFIs and the Blind signature, as is also done inside the `bench_all_bs.sh` script.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Integrating C/C++ primitives into a Rust environment via FFIs introduces potential memory concerns. While Rust provides a memory-safe environment for the protocol logic, the underlying building blocks remain subject to C/C++ memory management. We used Valgrind to find and mitigate memory issues occurring due to interaction between Rust and C/C++ codebase space. Our build scripts compile and link the components in our construction. These scripts are designed to be non-destructive and operate within the directories of our artifact. The build files for the FFIs for MAYO (`mayo-c-sys/build.rs`, `mayo-c-rain-sys/build.rs`) call a Makefile that recompiles MAYO with small additional changes. This process computes shared libraries in the respective folders (`mayo-c-sys`, `mayo-c-rain-sys`), which are cleaned up after being combined into a single shared library. Our benchmark script (`bench_all_bs.sh`) modifies the executability of the underlying scripts that build the respective modules in our code. First, the script cleans the build repository of the ZK-proofs (`vole/faest-cpp-tmp/build_debug`) and the build folders of the Blind signatures themselves (`<bs-name>/target`) before running the individual compilations and test executions that are then measured for performance. All executions are local, and we did not identify any ethical concerns relevant regarding the evaluation of this artifact.

A.2.2 How to access

The artifacts of this work are available here: https://github.com/shibammukherjee/pq_blind_signatures, permanent Zenodo concept DOI: https://doi.org/10.5281/zenodo.17979640 pointing to latest version and the Zenodo first version DOI: https://doi.org/10.5281/zenodo.17979641.

A.2.3 Hardware dependencies

We only tested our code on x86 architectures.

A.2.4 Software dependencies

We use and test only on Ubuntu 24.04. The main dependencies are inherited by the underlying building blocks we used. We iterate them, and in braces, we provide the version that we tested. The VOLEitH framework requires `gcc` (>= 14.2.0) or `Clang` (>= 19.1.7), `meson` (>= 1.7.0) and `ninja` (>= 1.12.1).

[1] https://github.com/PQCMayo/MAYO-C
[2] https://github.com/faest-sign/faest-avx
[3] https://github.com/rust-lang/rust-bindgen

The build of MAYO and the RainHash requires `cmake` (>= 3.28.3). The protocol level, i.e., Rust, requires `libclang` to be installed to parse the C/C++ headers for `bindgen`. It is possible to run supplementary tests and benchmarks for some of our constructions with `criterion`[4]. `Criterion` itself requires `gnuplot` for visualizations. These tests can be used to gain additional insights and more confidence in our claims (see Section A.5). For out-of-the-box benchmark runs, use our `Dockfile` built on Ubuntu 25.10, auto-loading all dependencies.

A.2.5 Benchmarks

Our Blind signature size and prove/verify runtime claims can be reproduced by running the `bench_all_bs.sh` script. The script benches each scheme for each security level (NIST Levels 1,3,5) and version (short, fast). Besides our script that runs all benchmarks together, we have scripts to only run the individual constructions: `bench_conservative_deg16_bs.sh`, `bench_optimized_bs.sh`, `bench_rainhash_bs.sh` and `bench_conservative_bs.sh`. The prove/verify runtime may vary depending on the hardware specification of the host machine.

A.3 Set-up

A.3.1 Installation

Installing all software dependencies and verifying their versions are sufficient. We have two options that allow you to verify our claims. You can either follow a native-build for which we provided a manual step-by-step guide explaining how the installation could work. We use a *fresh installation of WSL with Ubuntu 24.04* for this guide, so it follows the minimal steps. The guide can be found in our artifacts in the file: `manual-installation.md`. Alternatively, we provide a dockerfile `Dockerfile` that does the entire setup and the verification can be performed by running the underlying bench scripts. The instructions are also contained as comments in the dockerfile.

A.3.2 Basic Test

Finishing the installation, one runs the bench `bench_all_bs.sh` script. It is, however, possible to run one of the tests from the underlying explicit constructions for a smaller functionality test, e.g., `cd blind-signatures-conservative-deg16 && RUST_MIN_STACK=8388608 cargo test test_and_bench_sign_loop_conservative_128fv1`, which runs a test for SHAKE256-deg16+MAYO-128_f.

[4] https://docs.rs/criterion/latest/criterion/

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): On the 128-bit security level, each of our constructions outperforms current lattice-based proposals (see Table 1) with competitive or improved size. This is proven by our implementations described in Section 8 "Implementation" whose results are illustrated/reported in Table 3 and can be verified with the benchmark script.

A.4.2 Experiments

Our experiments are limited to evaluating the runtime of our algorithms.

(E1): [30 human-minutes[5] + 15 compute-minutes]: This test runs each algorithm described in Table 3 $((\mathsf{Sig}_i)_{i\in\{1,2,3\}}, \mathsf{Ver})$ and reports on their runtime in ms for each security level.
Preparation: Finish the installation as described in Section A.3.1.
Execution: Run the benchmark script `bench_all_bs.sh`.
Results: All of these results are written into the file `bench_log.txt`. The reported runtime numbers, and the signature sizes computed with the VOLEitH signature size formula and checked with total memory allocation, are then displayed in the file. The file has the same format as Table 3, listing first the construction, then the reported performance, and finally the reported sizes of the Blind signatures.

A.5 Notes on Reusability

We have provided some additional benchmarking options using `criterion`. These can also be used for benchmarking with more detailed statistics. We provide FFIs for optimized codebases written in C/C++, which can be reused as they are. These are suffixed with the term `-sys` as is standard for FFIs in Rust. The VOLEitH proof for SHAKE256 and RainHash are embedded in `vole/conservative_bs/owf_proof.inc`. Our ZK-proof use a shared codebase that can be modified to implement other circuit constructions or reuse the same for other additional ZK-proofs over involving, for example, SHAKE256. Similarly, it is also possible to modify the proof of SHAKE256, to, for example, support TurboSHAKE256.

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/.

[5] This primarily depends on the installation time, which should hopefully be relatively fast with the dockerfile and the manual installation guide