USENIX Security '24 Artifact Appendix: SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange

Phillip Gajland[1,2], Bor de Kock[3], Miguel Quaresma[1], Giulio Malavolta[4,1], and Peter Schwabe[1,5]

[1]Max Planck Institute for Security and Privacy, Bochum, Germany
[2]Ruhr University Bochum, Bochum, Germany
[3]NTNU – Norwegian University of Science and Technology, Trondheim, Norway
[4]Bocconi University, Milan, Italy
[5]Radboud University, Nijmegen, The Netherlands
phillip.gajland@{mpi-sp.org,rub.de}, bor.dekock@ntnu.no,
{miguel.quaresma,giulio.malavolta}@mpi-sp.org, peter@cryptojedi.org

A Artifact Appendix

A.1 Abstract

This is the artifact for the "Swoosh: Efficient Lattice-Based Non-Interactive Key Exchange" paper.

Although earlier work has shown that lattice-based Non-Interactive Key Exchange (NIKE) is theoretically possible, it has been considered too inefficient for real-life applications. In this work, we challenge this folklore belief and provide the first evidence against it. We construct an efficient lattice-based NIKE whose security is based on the standard module learning with errors (M-LWE) problem in the quantum random oracle model.

To substantiate our efficiency claim, we provide an optimised implementation of our passively-secure construction in Rust.

Our scheme achieves a post-quantum security level exceeding 120 bits.

The main contributions of this artifact are:

- Optimized implememtation of Passive-Swoosh, a Lattice-based Non-Interactive Key Exchange
- Sage script to estimate security of Passive-Swoosh

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact contents of the are accessible at: https://git.noc.ruhr-uni-bochum.de/miranmfx/swooshuae/-/tree/477a06ba19b84ccf5e0994bed0a2e20de97acb87, with commit hash `477a06ba19b84ccf5e0994bed0a2e20de97acb87`.

A.2.3 Hardware dependencies

No special hardware is required to evaluate the artifact aside from a machine running an AMD64 CPU.

A.2.4 Software dependencies

In order to evaluate the artifact and reproduce the claims made, an installation of the Rust compiler, version 1.62.18, is required. Furthermore, the Sage CAS, should also be installed in order to run replicate the security estimates that underly the claims made in the paper.

A.2.5 Benchmarks

See below.

A.3 Set-up

The artifact can be set up via Docker or manually. Instructions for both set-ups are provided below.

A.3.1 Docker set-up

This artifact contains a Dockerfile which sets up a container with Rust and Sage along with the contents of the artifact. For information on the installation of Docker see https://docs.docker.com/get-docker/.

To setup Rust and Sage using Docker run:

```
docker build -t swoosh .
```

You can run the Docker images with:

```
docker run -it swoosh
```

Remarks: please note that running benchmarks inside a container will impact the measurements.

A.3.2 Manual set-up

To install Rust & Sage run the following commands in the artifact directory:

```
# 1. Install Sage
## On Debian and derivatives
apt-get install sagemath python3 python3-pip

## on macOS
brew install sage

# 2. Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs
↪  | sh

# 3. Install Python dependencies (required by
↪  lattice-estimator)
pip install -r security/requirements.txt
```

For more information on Rust installation see: https://www.rust-lang.org/tools/install.
For more information on Sage installation see: https://doc.sagemath.org/html/en/installation.

Remarks: please note that Rust should be at least on verion 1.59. For this work, Rust version 1.62 was used.

A.3.3 Testing the set-up

A basic sanity check to ensure the code provided works as expected can be done using one of two methods. The following command will run the unit tests for the scheme:

```
make test
```

This command should output the following[1]:

```
Running unittests src/lib.rs
↪  (target/debug/deps/ref0-c35ef30ce0a779e4)

running 1 test
test tests::test_scheme ... ok

successes:

successes:
    tests::test_scheme

test result: ok. 1 passed; 0 failed; 0 ignored; 0
↪  measured; 15 filtered out; finished in 0.10s
```

To run a basic correctness test on the scheme, use the following command:

```
make test_scheme
```

This command should output the following[2]:

```
Running unittests src/lib.rs
↪  (target/debug/deps/ref0-c35ef30ce0a779e4)

running 16 tests
test arithmetic::fq::tests::test_cmp ... ok
test arithmetic::fq::tests::test_fp_add ... ok
test arithmetic::fq::tests::test_fp_bytes ... ok
test arithmetic::fq::tests::test_fp_mul ... ok
test arithmetic::fq::tests::test_fp_sub ... ok
test arithmetic::poly::tests::test_poly_add ... ok
test arithmetic::poly::tests::test_poly_bytes ... ok
test arithmetic::poly::tests::test_poly_ntt ... ok
test arithmetic::polyvec::tests::test_polyvec_add ... ok
test arithmetic::polyvec::tests::test_polyvec_bytes ...
↪  ok
test tests::test_getnoise ... ok
test tests::test_rec ... ok
test tests::test_round ... ok
test arithmetic::poly::tests::test_poly_basemul ... ok
test tests::test_genoffset ... ok
test tests::test_scheme ... ok

successes:

successes:
    arithmetic::fq::tests::test_cmp
    arithmetic::fq::tests::test_fp_add
    arithmetic::fq::tests::test_fp_bytes
    arithmetic::fq::tests::test_fp_mul
    arithmetic::fq::tests::test_fp_sub
    arithmetic::poly::tests::test_poly_add
    arithmetic::poly::tests::test_poly_basemul
    arithmetic::poly::tests::test_poly_bytes
    arithmetic::poly::tests::test_poly_ntt
    arithmetic::polyvec::tests::test_polyvec_add
    arithmetic::polyvec::tests::test_polyvec_bytes
    tests::test_genoffset
    tests::test_getnoise
    tests::test_rec
    tests::test_round
    tests::test_scheme

test result: ok. 16 passed; 0 failed; 0 ignored; 0
↪  measured; 0 filtered out; finished in 0.10s
```

A.4 Evaluation workflow

The major claims and experiments described in the paper were obtained on an **Intel Core i7-6500U (Skylake)** running on a single core with **Hyper-threading and TurboBoost disabled**. The Rust compiler version used for the benchmarks was **1.62.18**.

A.4.1 Major Claims

(C1): Passive-Swoosh shared-key derivation is faster, by a factor of 48, than CTIDH, another post-quantum NIKE. Key generation is faster by a factor of ≈ 3.

(C2): Compared to post-quantum KEMs, such as Kyber-768 and mceliece348864, Passive-Swoosh is several orders of magnitude slower.

[1] The warnings about the external AES implementation can be ignored.

[2] Same as before.

(C3): Swoosh achieves over 120 bits of post-quantum security.

A.4.2 Experiments

(E1): Passive Swoosh cycle counts: obtain the cycle count measurements for Passive-Swoosh presented in Table 2 of the original paper.
How to: Run the following command in the artifact's root directory

```
make bench_scheme
```

This command should output the following[3]:

```
Running `target/debug/bench_scheme`
keygen (cycles):
average: 80062472
median: 89441519

skey_deriv (cycles):
average: 5522018
median: 6212299
```

(E2): Passive Swoosh security level: obtain an estimate for the post-quantum security level achieved by the scheme.
How to: Run the following command in the artifact's root directory

```
cd security && sage swoosh_estimator.py
```

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/usenixsec2024/.

[3] Same as before.