USENIX Security '26 Artifact Appendix: mmCipher: Batching Post-Quantum Public Key Encryption Made Bandwidth-Optimal

Hongxiao Wang
The University of Hong Kong

Ron Steinfeld
Monash University

Markku-Juhani O. Saarinen
Tampere University

Muhammed F. Esgin
Monash University

Siu-Ming Yiu
The University of Hong Kong

A Artifact Appendix

A.1 Abstract

We provide self-contained Python and portable C implementations of the scheme (with some aspects that may need further optimization for production-level deployment). The artifact also contains the code used for generating the comparative benchmarks reported in this work, source code for ZK proof experiments using the LaZer Library, and scripts and tools used for security parameter selection (computation of lattice parameter sets and decryption failure probabilities).

A.2 Description & Requirements

This is a software artifact that has been developed and tested on Ubuntu Linux 24.04 LTE operating system. Running on other UNIX-like operating systems and MacOS should also be straightforward. All external software and library dependencies are standard open-source components.

The artifact contains six directories, each with a separate README file containing running instructions.

Subdirectory	Description
mmCipher-c	Plain C Implementation of mmCipher. Used for benchmarks.
mmCipher-py	Python model for mmCipher, compatible with the C code.
mmCipher-pkzk	ZK proofs of public key validity with LaZer.
refKyber-c	Plain C Kyber ref code for "apples-to-apples" benchmarks.
pr-fail-dec	Computation of decryption/decaps failure probabilities.
param-sage	SageMath lattice parameter selection/exploration scripts.

Note: Only **mmCipher-c** is related to our major claims (of good performance and bandwidth-optimality); other directories serve as "nothing-up-the-sleeve" research artifacts that allow the parameter selection and other aspects of the design to be independently reproduced and validated.

We are including the **mmCipher-pkzk** artifact (for public-key validity proofs) despite it being somewhat difficult to use, as the Lazer library it depends on is in an unstable state. These zero-knowledge proofs are not required during the regular operation of mmCipher and hence do not impact the main claims.

A.2.1 Security, privacy, and ethical concerns

The purpose of the artifact is to reproduce benchmarks and specific mathematical computations related to the mmCipher paper. This software only requires user-level privileges to execute. The software does not interact with other local or remote files (outside its own working directory). Hence, the artifact should not present security, privacy, or ethical concerns.

A.2.2 How to access

The artifact files have been archived as `https://doi.org/10.5281/zenodo.17849532`. The same files may also be accessed as a GitHub repository: `https://github.com/ml-kem/mmcipher-artifact`.

A.2.3 Hardware dependencies

No specific hardware requirements; a standard x86 or ARM workstation/laptop can be used.

A.2.4 Software dependencies

Development was conducted on Linux and MacOS systems; most UNIX-like operating systems should work. The artifact has been tested on Ubuntu 24.04 LTE.

See README files in each subdirectory for a more detailed description:

- **mmCipher-c** (and others): C compiler (gcc or clang) toolchain and build tools (e.g., make).
- **mmCipher-py**: Python 3.11+, with the `pycryptodome` package (`https://pypi.org/project/pycryptodome/`)

- **mmCipher-pkzk**: Uses the lazer library: https://github.com/lazer-crypto/lazer
- **refKyber-c**: C build tools, Python, and bash scripting. To re-create the plots and tables in the paper from performance data (`plot/make_plots.sh`), Python `matplotlib` and `scipy` are required.
- **pr-fail-dec**: GNU Multiprecision (GMP) library (Ubuntu/Debian packages libgmp-dev and libmpfr-dev).
- **param-sage**: Sage is required to run the scripts (tested with SageMath 10.7). https://www.sagemath.org/

A.2.5 Benchmarks

There are no external benchmark requirements.

A.3 Set-up

We will focus on the basic performance and bandwidth benchmarks contained in the `mmCipher-c` subdirectory.

A.3.1 Installation

The C implementation only requires a suitable C compiler toolchain and build tools such as `make`. The exact method of obtaining cycle counts is non-portable; cycle counter access is provided on x86 platforms.

A.3.2 Basic Test

Invoking `make test` performs a test-vector-based functional test on all parameter sets:

```
$ cd mmCipher-c
$ make test
```

A successful run concludes with:

```
ea0510a559aa66e5bbc0859f2544bf7231823ad37606
e892f8865421ba5243bc  testvec.tmp
cmp testvec.tmp testvec.txt
```

The process will stop at compilation errors, and the final file comparison (and the test vector hash) will not match if the implementation does not match with the test vectors.

A.4 Evaluation workflow

The `mmCipher-c` implementation provides facilities for performance testing and functional testing, depending on the TEST variable passed on to the Makefile. The test program created by the Makefile is named `xtest`. To change its behavior, please see the options in the Makefile. For example, to create and run a checksum-generating executable for the 192-bit PKE parameter set:

```
$ make MODE=PKE LEVEL=192 TEST=TESTVEC
$ ./xtest
```

You will have to `make clean` when changing from one parameter set to another, since the parameters are defined as macros.

There are some known-good checksums in `testvec.txt`, in a format also produced by the Python implementation (in the directory `mmCipher-py`). The test vectors consist of lines such as

```
sk[208] chk 8baed64e
```

Where `sk` is the variable (here, secret key), followed by the byte length (here 208), and a 32-bit CRC checksum of the data item.

A.4.1 Major Claims

A multi-message multi-recipient PKE/KEM (mmCipher) enables the batch encryption of multiple messages (as a message vector) for multiple independent recipients in a single operation, significantly reducing costs, particularly bandwidth, compared to the trivial solution of encrypting each message individually using a standard PKE/KEM such as FIPS 203 ML-KEM a.k.a. Kyber.

(C1): The bandwidth requirement of mmCipher is asymptotically optimal: The ciphertext size divided by the number of recipients N approaches a constant as N grows: Figure 7 (and Table 9) in the paper.

(C2): The performance of mmCipher is comparable (or better) to ML-KEM at equivalent security levels: Figure 8 (and Table 8) in the paper.

Note: While we claim sufficient cryptanalytic security (when mmCipher is used appropriately), we do not claim that this implementation is secure against side-channel attacks. Furthermore, some components (especially Gaussian samplers) in the present implementation can be considered as placeholders and must be improved for use in a production environment (which requires further research). More specifically, this code is **not** claimed to be fully constant-time, although an attempt has been made in some segments of the code. However, we expect a "production" implementation to have similar performance characteristics (or better, as the present implementation is in plain C and lacks assembler optimizations).

A.4.2 Experiments

(E1): [60 human-minutes + 15 compute-minutes]:
How to: The basic experiment is to run the mmCipher-C implementation benchmarks; observe the reported ciphertext sizes (Claim C1) and performance metrics (Claim C2).
For performance comparison to ML-KEM, a reference C implementation is reproduced in the `refKyber-c` directory.

Preparation: Optional: Dynamic frequency scaling ("Turbo Boost") and related cache effects can significantly affect reproducibility of performance measurements. A standard practice is to disable it.

Intel-based Linux system: `echo 1 > /sys\ /devices/system/cpu/intel_pstate/no_turbo`

AMD-based Linux system: `echo 0 > /sys\ /devices/system/cpu/cpufreq/boost`

Execution: Move to directory `mmCipher-c` in the artifact. A file `bench.txt` is produced by running `make bench`. This takes a few minutes. See the file `bench-mm.txt` for the data that was used in the paper.
To obtain the ML-KEM (Kyber) comparison data, move to `refKyber-c` directory and run the shell script `./run_bench.sh`. This produces an *another* file named `bench.txt`. See the file `bench-ref.txt` for the data that was used in the paper.

Results: An mmCipher-c benchmark run produces a file `bench.txt` where each non-comment line has: The algorithm parameter set (e.g. mmCipher-PKE-128), the item being measured (e.g. mmSetUp), the number of recipients (e.g. N=1024), and alternatively a bandwidth byte size (e.g. len=) or the timing measurements (cyc= cycle count, sec= wall-clock timing). Note that running on ARM targets will not produce cycle counts; only wall-clock timings will be reported. Secret key and public key sizes can be read from the test vector data file `testvec.txt` (or the generated file `testvec.tmp`).
For comparison purposes, run refKyber-c on the same platform. This produces a `bench.txt` file in a format similar to the produced by mmCipher-c, but also containing parameter data.
One can manually interpret the benchmark results, e.g., by dividing the ciphertext size by the number of recipients (Claim C1) or by comparing the benchmark measurements between mmCipher and Kyber (Claim C2).
Some ad hoc analytic scripts are provided in the `refKyber-c/plot` directory. Running the shell script `make_plots.sh` in that directory recreates files `bytes_relative.pdf` (Figure 7) and `speed_relative.pdf` (Figure 8) of the paper from the data files `../bench-ref.txt` and `../../mmCipher-c/bench-mm.txt`. Different data files can be substituted here if desired.

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/.