ISSN 1975-4736 ⓒMITA2024

Enhancing ML-KEM Performance through Macro-Based Modular Reduction Optimization

Mooh Ludivine Esther Makafui
Department of Information Security
Pukyong National University
Busan, South Korea
ludivinemooh@pukyong.ac.kr

Ki-Ryong Kwon, Piljoo Choi
Division of Computer Engineering and AI
Pukyong National University
Busan, South Korea
pjchoi@pknu.ac.kr

***Abstract*- This paper presents an optimization technique to enhance the performance of ML-KEM, a lattice-based key encapsulation mechanism designed for post-quantum security. This technique converts the modular reduction functions into macros, eliminating the overhead associated with function calls. This optimization addresses performance bottlenecks in ML-KEM, resulting in substantial improvements in execution time: a 25.30% reduction for key generation, a 20.05% improvement for encapsulation, and a 28.70% reduction for decapsulation compared to the reference implementation. By enhancing the efficiency of modular reduction, this work contributes to the practicality and potential widespread adoption of post-quantum cryptography in real-world scenarios.**

Keywords—ML-KEM, lattice-based cryptography, post-quantum security, number-theoretic transform, modular reduction, performance optimization, cryptographic algorithms, computational efficiency

I. INTRODUCTION

ML-KEM, a lattice-based key encapsulation mechanism, has gained recognition for its security and performance in the realm of post-quantum cryptography [3], [5]. However, the practical deployment of ML-KEM faces challenges regarding computational efficiency, particularly in resource-constrained environments.

Modular reduction is a critical component of ML-KEM that ensures the results of arithmetic operations remain within the defined finite field [3], [2]. Optimizing modular reduction is crucial for improving the overall performance of ML-KEM . In this paper, we propose converting the Montgomery reduction and Barrett reduction functions [2], [1] into macros to enhance the efficiency of modular reduction in ML-KEM.

Our work focuses on low-level optimizations that are applicable across different target devices, aiming to provide a clean and efficient implementation of ML-KEM without being tied to specific hardware architectures. By improving the computational efficiency of ML-KEM, our work addresses the performance challenges associated with lattice-based cryptography and contributes to the practicality and potential widespread adoption of post-quantum cryptography in real-world scenarios.

II. PROPOSED OPTIMIZATIONS

A. Converting Modular Reduction Functions to Macros

Our optimization technique involves converting Montgomery reduction and Barrett reduction [2], into macros. Montgomery reduction transforms coefficients into a Montgomery representation and performs reduction using efficient bit-shifting operations. We changed Montgomery reduction in the following equation from a function to a macro.

$$t = (a + (a \cdot Q' \bmod R) \cdot Q)/R,$$

where $Q' \times Q \equiv -1 \pmod{R}$.

Similarly, Barrett reduction [1] is another technique for efficient modular reduction. Barrett reduction approximates the quotient of the coefficient divided by the modulus using precomputed constants. We also changed Barrett reduction in the following equation from a function to a macro.

$$t = a - \left(\left\lfloor \frac{a \cdot \mu}{2^k} \right\rfloor\right) \cdot Q$$

The macro-based optimizations eliminate the function call overhead, variable declarations, and the need for explicit loads and stores, computing the reduction directly using the input value and precomputed constants.

B. Instruction Count Analysis

To demonstrate the efficiency of our macro-based optimizations, we analyze the instruction counts of the original functions and the optimized macros. Table 1 provides a detailed breakdown of the instruction counts, including fetching, loading, operations, and returning overhead.

TABLE I. INSTRUCTIONS COUNTS MONTGOMERY REDUCTION

Computation		Original Code	Proposed Optimized Code
Reduction		7	7
Overhead	Calling	5	0
	Returning	2	0
Total		14	7

TABLE II. INSTRUCTIONS COUNTS BARRETT REDUCTION

Computation		Original Code	Proposed Optimized Code
Reduction		8	8
Overhead	Calling	5	0
	Returning	3	0
Total		16	8

As shown in Tables 1 and 2, the macro-based optimizations significantly reduce the total instruction count by eliminating the function calling and returning overhead. By converting the modular reduction functions into macros, we eliminate the need for function prologue and epilogue instructions, which include setting up the stack frame, saving registers, and restoring registers upon function return. This optimization results in a more streamlined and efficient execution of the modular reduction operations.

The reduction in instructions count directly translates to improved performance, as fewer instructions need to be fetched, decoded, and executed by the processor. This optimization is particularly beneficial in resource-constrained environments where minimizing the number of executed instructions is crucial for achieving high performance. However, it is important to note that the code size increases from 45,093 bytes to 46,221 bytes due to the inline expansion of the macro. When using functions for Montgomery and Barrett reduction, the code for the reduction operations is stored separately in memory. Each time the function is called, the program needs to jump to the function's memory location, execute the instructions, and then return to the calling point. This process introduces overhead in terms of function call setup, parameter passing, and return value handling. The total latency for a function includes the reduction operations (7 instructions for Montgomery, 8 for Barrett) and the function calling and returning overhead (7 instructions for Montgomery, 8 for Barrett).On the other hand, when the reduction functions are converted to macros, the compiler expands the macro inline at each usage site. This means that the reduction code is directly inserted into the program flow, eliminating the need for function calls and the associated overhead. The macro expansion results in a larger code size because the reduction code is repeated at each usage site. However, this increase in code size is offset by the significant reduction in latency. The macro-based implementation only includes the reduction operations (7 instructions for Montgomery, 8 for Barrett), without any additional overhead for function calls and returns.

III. PERFORMANCE EVALUATION

A. Implementation and set up

We implemented the proposed optimizations using the ML-KEM reference code obtained from the ML-KEM repository [4]. The optimization was applied directly to the C code, focusing on the key generation, encapsulation, and decapsulation algorithms. All benchmarking and testing were conducted on a virtual machine running Ubuntu on Windows Subsystem for Linux (WSL) on an Intel Core i7-1165G7 processor with a clock speed of 2.1 GHz. We compared the performance of three different implementations of ML-KEM.

TABLE III. IMPLEMENTATION FEATURES

Implementation	Feature
Reference	reference implementation submitted by the Kyber (ML-KEM) team to NIST, serving as a baseline for comparison.
This work	Our proposed optimization, which incorporates the macro-based modular reduction optimization discussed in this paper. This optimization is applied directly to the C code of the reference implementation.
libcrux [7]	libcrux implementation of ML-KEM/Kyber, using rust and which includes signed representatives stored in wider i32s instead of i16s.

To evaluate the performance of our optimizations, the execution times were measured using the built-in benchmarking functionality provided in the original repository [4]. We compare the performance of this work with the reference implementation to assess the impact of our macro-based modular reduction optimization. Furthermore, we compare our results with the libcrux implementation to show the effectiveness of our approach.This comparison highlights the potential of our optimization technique,demonstrating its ability to achieve significant performance gains.

B. Results and Analysis

The performance evaluation results demonstrate significant improvements in execution time achieved by our proposed optimizations. Table 4 presents the average cycle counts for key generation, encapsulation, and decapsulation, comparing the reference implementation [4], our proposed optimization implementation, and the LibCrux implementation [7].

TABLE IV. LATENCY IN CLOCK CYCLES AND CODE SIZE

Operation	This work	Reference	LibCrux
Key Pair (CCs)	44,446	59,505	51,859
Encaps (CCs)	49,147	61,466	59,795
Decaps (CCs)	47,364	66,434	63,480
Code size (bytes)	46,221	45,093	N/A

As shown in Table 4, our optimized implementation achieves a 25.30% reduction in execution time for key generation, a 20.05% improvement for encapsulation, and a 28.70% reduction for decapsulation compared to the reference implementation. These improvements can be attributed to the conversion of modular reduction functions to macros.

It is worth noting that the proposed optimizations do not compromise the security of ML-KEM [6]. The optimizations are applied at the implementation level and do not alter the underlying mathematical structure or security assumptions of the scheme. While our optimizations do not inherently compromise the security of ML-KEM, as they do not alter the

underlying mathematical structure or security assumptions, it is important to perform rigorous security analysis testing to ensure the implementations remain secure against potential vulnerabilities and side-channel attacks.

IV. CONCLUSION

In this paper, we presented the optimization of converting modular reduction functions to macros, which eliminates the overhead of function calls and streamlines the execution of modular reduction operations. Our optimized implementation, benchmarked on an Intel Core i7-1165G7 processor, demonstrates significant improvements in execution time compared to the reference implementation. The optimizations achieve a 25.30% reduction in execution time for key generation, a 20.05% improvement for encapsulation, and a 28.70% reduction for decapsulation. By improving the computational efficiency of ML-KEM, our work addresses the performance challenges associated with lattice-based cryptography.In conclusion, our work presents significant performance optimizations for ML-KEM from a software perspective, contributing to the advancement of post-quantum cryptography. By focusing on software-level optimizations that are applicable across different target devices, we aim to provide a clean and efficient implementation of ML-KEM. Future research efforts should focus on further enhancing the proposed optimization techniques, conducting thorough security evaluations, and promoting the integration of optimized ML-KEM into software libraries and frameworks.

Acknowledgement

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-2020-0-01797) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

References

[1] P. Barrett, “Implementing Barrett’s reduction,” 1986.

[2] G. Seiler, "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography," Cryptology ePrint Archive, Report 2018/039, 2018. [Online]. Available: https://eprint.iacr.org/2018/039.pdf.

[3] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS-Kyber: Algorithm specifications and supporting documentation,” 2021.

[4] "Kyber key encapsulation mechanism" GitHub repository. [Online]. Available: https://github.com/pq-crystals/kyber.

[5] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, “Post-quantum key exchange for the TLS protocol from the ring learning with errors problem,” 2020.

[6] Yiqiang Zhao, Shijian Pan, Haocheng Ma, Ya Gao, Xintong Song, Jiaji He, Yier Jin, "Side Channel Security Oriented Evaluation and Protection on Hardware Implementations of Kyber," IEEE Transactions on Circuits and Systems I: Regular Papers, 2023.

[7] "LibCrux: Kyber Implementation," GitHub repository. [Online]. Available:https://github.com/cryspen/libcrux/tree/main/src/kem/kyber

[8] K. Bhargavan, F. Kiefer, and G. Tamvada, "Verified ML-KEM (Kyber) in Rust," January 16, 2024.