Computer Systems Science & Engineering
DOI: 10.32604/csse.2023.035064
Article

Tech Science Press

Portable and Efficient Implementation of CRYSTALS-Kyber Based on WebAssembly

Seog Chung Seo[1] and HeeSeok Kim[2,*]

[1]Department of Financial Information Security, Kookmin University, Seoul, 02707, Korea
[2]Department of Cyber Security, College of Science and Technology, Korea University, Sejong, 30019, Korea
*Corresponding Author: HeeSeok Kim. Email: 80khs@korea.ac.kr
Received: 05 August 2022; Accepted: 11 November 2022

Abstract: With the rapid development of quantum computers capable of realizing Shor's algorithm, existing public key-based algorithms face a significant security risk. Crystals-Kyber has been selected as the only key encapsulation mechanism (KEM) algorithm in the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) competition. In this study, we present a portable and efficient implementation of a Crystals-Kyber post-quantum KEM based on WebAssembly (Wasm), a recently released portable execution framework for high-performance web applications. Until now, most Kyber implementations have been developed with native programming languages such as C and Assembly. Although there are a few previous Kyber implementations based on JavaScript for portability, their performance is significantly lower than that of implementations based on native programming languages. Therefore, it is necessary to develop a portable and efficient Kyber implementation to secure web applications in the quantum computing era. Our Kyber software is based on JavaScript and Wasm to provide portability and efficiency while ensuring quantum security. Namely, the overall software is written in JavaScript, and the performance core parts (secure hash algorithm-3-based operations and polynomial multiplication) are written in Wasm. Furthermore, we parallelize the number theoretic transform (NTT)-based polynomial multiplication using single instruction multiple data (SIMD) functionality, which is available in Wasm. The three steps in the NTT-based polynomial multiplication have been parallelized with Wasm SIMD intrinsic functions. Our software outperforms the latest reference implementation of Kyber developed in JavaScript by ×4.02 (resp. ×4.32 and ×4.1), ×3.42 (resp. ×3.52 and ×3.44), and ×3.41 (resp. ×3.44 and ×3.38) in terms of key generation, encapsulation, and decapsulation on Google Chrome (resp. Firefox, and Microsoft Edge). As far as we know, this is the first software implementation of Kyber with Wasm technology in the web environment.

Keywords: Crystals-kyber; post-quantum cryptosystem (PQC); javascript; WebAssembly; SIMD; web application; internet of things (IoT); edge computing

1 Introduction

As the Internet of Things (IoT) era dawns, several heterogeneous devices are now connected to the Internet, and they need to protect transmitted secret information or privacy-sensitive data. Typically, an encryption algorithm such as advanced encryption standard (AES), a block cipher, is used to protect transmitted data from eavesdroppers. To share the key, used for encryption and decryption, between a transmitter and a receiver, current public key-based key transport (e.g., Rivest–Shamir–Adleman (RSA)-based key transport) or key agreement algorithm (e.g., Diffie–Hellman (DH) and elliptic curve DH (ECDH)) are widely used. However, with the rapid advancement of quantum computers capable of realizing Shor's algorithm, currently used public key-based keying algorithms will no longer be secure. Thus, currently, National Institute of Standards and Technology (NIST) is conducting a competition for post-quantum cryptography (PQC) standard that is secure against quantum computers. NIST requires two types of PQC: quantum-resistant key encapsulation mechanism (KEM) for key establishment and quantum-resistant digital signature [1]. In July 2022, NIST selected the final standard algorithms: Crystals-Kyber [2] for KEM and Crystals-Dilithium [3], Falcon [4], and Sphincs+ [5] for Digital Signature Algorithm (DSA) and announced four round candidates. The reason for selecting Crystals-Kyber is its efficiency in terms of computation and the size of ciphertext and keys. Because Crystals-Kyber is the only algorithm for KEM standards, optimizing its performance in the web environment is crucial. As aforementioned, because many heterogeneous devices are used in the IoT environment, it is required to develop IoT applications that can efficiently run on any device. Therefore, the portability of IoT applications is becoming more important. Web browsers are the most representative platform for web applications running on IoT devices, and JavaScript, a cross-platform/browser language, is currently the most popular language for portable web applications. Until now, some studies have been conducted for developing portable Lattice-based PQC libraries based on Javascript for secure communication in the IoT web environment [6–8]. Although Javascript-based crypto libraries can be executed on any IoT device due to their portability, their performance is significantly lower than software based on native languages such as C/C++ and Assembly.

WebAssembly (Wasm), released in 2017, is a portable execution framework for developing high-performance web applications [9,10]. Wasm enables C/C++-based native codes to run on web applications. Namely, Wasm modules implemented with native languages can be loaded and used by Javascript applications in the web environment. Furthermore, with Wasm, multiple data of the same type can be processed with single instruction multiple data (SIMD) instructions. Recently, BoSun Park et al. presented a crypto library based on JavaScript and Wasm to provide portable, efficient, and secure communication in web-based IoT systems [11]. However, to date, there is no Wasm-based PQC library; thus, this study presents an optimized Crystals-Kyber-KEM library based on Wasm for both efficiency and portability in the web environment.

The contribution of this study can be summarized as follows.

- **Optimizing NTT-based Polynomial Multiplication using SIMD instructions available on Wasm Framework**

We optimize number theoretic transform (NTT)-based polynomial multiplication, one of the most core operations in the Kyber algorithm, with SIMD intrinsic instructions available in Wasm. With a 128-bit Wasm SIMD register, it is possible to process eight coefficients because q is 3329, which is less than 2^{12}. Our parallel implementations of NTT conversion, point-wise multiplication, and inverse NTT conversion provide ×6.83, ×4.95, and ×7.99 of improved performance compared with the latest Javascript implementation from [8]. As far as we know, this is the first work optimizing NTT-based polynomial multiplication with Wasm SIMD functionality.

- **Presenting Efficient and Portable Crystals-Kyber Library based on Wasm Framework**

We present the first optimized Crystals-Kyber library by fully utilizing the Wasm framework for efficient and portable secure communication in the web environment. We construct our Wasm module containing an NTT-based polynomial multiplication function, which is the most time-consuming operation, except for the secure has algorithm (SHA)-3-based hash function in Kyber. The rest of the Kyber operations are implemented with Javascript. Thus, our Kyber library is also an efficient and portable PQC KEM library in the web environment. On Google Chrome (resp. Firefox and Microsoft Edge), our Kyber library provides approximately ×4.02 (resp. ×4.32 and ×4.1), ×3.42 (resp. ×3.52 and ×3.44), and ×3.41 (resp. ×3.44 and ×3.38) better performance than the latest Javascript Kyber implementation [8] in terms of key generation, encapsulation, and decapsulation, respectively.

- **Presenting Optimized Wasm module implementation for Kyber-768 parameter for Representative three Browsers**

We focus on optimizing Kyber-768, providing NIST security level 3. We measure the performance on three representative web browsers: Google Chrome, Firefox, and Microsoft Edge. Our experimental results on these three browsers can be useful when developing secure web applications using Crystals-Kyber on the three browsers.

The remainder of this paper is organized as follows. Section 2 describes some preliminaries about Crystals-Kyber, NTT-based polynomial multiplication, Kyber parameters, and Wasm. Section 3 describes previous Kyber implementations in the web environment. Section 4 presents the proposed parallel implementation methods for NTT-based polynomial multiplication. The experimental results are given in Section 5, and the conclusion is drawn in Section 6.

2 Preliminaries

This section describes the preliminary background of our works: the introduction of Kyber, NTT-based polynomial multiplication, Kyber parameters, Kyber operations, and Wasm/SIMD.

2.1 Crystals-Kyber

Crystals-Kyber (Kyber) is an indistinguishability under chosen-ciphertext attack-secure (IND-CCA2) KEM, and its security is based on the Module Ring-LWE problem [2]. The Kyber-KEM is constructed using the Kyber public key encryption (Kyber-PKE) using the Fujisaki–Okamoto (FO) transform. Kyber-PKE consists of three algorithms: key generation, encryption, and decryption. Except for the random sampling based on the hash function, the core operation of each Kyber-PKE algorithm is either matrix by vector multiplication or vector by vector multiplication. Note that each element of the matrix and vector is over Ring $R_q = \mathrm{Z}_q[X]/(X^n + 1)$, where n and q are 256 and 3329, respectively. In the following algorithm description, bold upper-case letters (e.g., $\mathbf{A}$) and bold lower-case letters (e.g., $\mathbf{s}$) denote matrixes and vectors, respectively.

2.2 NTT-based Polynomial Multiplication

The major operation of matrix by vector multiplication and vector by vector multiplication is the polynomial multiplication over R_q. The naive polynomial multiplication method requires $O(n^2)$ complexity, where n is 256 in the case of Kyber. To accelerate the speed of polynomial multiplication over R_q, the NTT-based polynomial multiplication method is widely used in the context of LWE-based cryptosystems, including Kyber, because its complexity is $\mathrm{O}(n\log n)$ [12,13]. NTT-based polynomial multiplication consists of three steps: NTT conversion, component-wise (or point-wise) multiplication, and inverse NTT conversion. In other words, the multiplication of two polynomials $f = \sum_{i=0}^{255} f_i X^i$ and $g = \sum_{i=0}^{255} g_i X^i$ in R_q is conducted in three steps.

- Converting each polynomial into the NTT domain with Eq. (1):

$$\hat{f} \leftarrow NTT(f) \text{ and } \hat{g} \leftarrow NTT(g), \text{ where } NTT(f) = \left(\widehat{f_0} + \widehat{f_1}X,\ \widehat{f_0} + \widehat{f_3}X,\ \ldots,\ \widehat{f_{254}} + \widehat{f_{255}}X\right)$$
$$\text{with } \widehat{f_{2i}} = \sum_{j=0}^{127} f_{2j}\zeta^{(2i+1)j} \text{ and } \widehat{f_{2i+1}} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(2i+1)j}\ (\zeta \text{ is the } 256 - \text{root of unity and } \zeta^{(2i+1)j} \text{ values are precomputed.}) \tag{1}$$

- Conducting component-wise multiplication with Eq. (2):

$$\hat{h} = \hat{f} \circ \hat{g} \text{ where } \widehat{h_{2i}} + \widehat{h_{2i+1}}X = \left(\widehat{f_{2i}} + \widehat{f_{2i+1}}X\right)\left(\widehat{g_{2i}} + \widehat{g_{2i+1}}X\right) \bmod \left(X^2 - \zeta^{2i+1}\right) \text{ for } i = 0, \ldots, 127. \tag{2}$$

- Applying inverse NTT conversion with Eq. (3):

$$fg = NTT^{-1}\left(\hat{h}\right) \tag{3}$$

For NTT conversion and inverse NTT conversion, typically the Cooley–Tukey (CT) method [14] and Gentleman–Sande (GS) method [15] are used, respectively.

2.3 *Kyber Parameters*

Table 1 shows the parameters of the Kyber algorithm. The degree n and prime q are fixed as 256 and 3329, respectively, among all security levels. k represents a security scaling factor, and it is the dimension of a matrix and vector such as $\mathbf{A} \in R_q^{k\times k}$ and $\mathbf{s} \in R_q^k$. η_1 , η_2, d_u, and d_v represent parameters to balance between the ciphertext size and failure probability in the process of central binomial distribution (CBDη), Compress, and Decompress. Kyber provides parameters for security levels 1, 3, and 5, which correspond to AES-128, AES-192, and AES-256 security levels, respectively.

Table 1: Crystals-kyber parameters (*SL* denotes the security level)

Parameters	NIST *SL*	n	q	k	$(\eta_1,\ \eta_2)$	$(d_u,\ d_v)$
Kyber-512	1 (AES-128)	256	3329	2	(3, 2)	(10, 3)
Kyber-768	3 (AES-192)	256	3329	3	(2, 2)	(10, 4)
Kyber-1024	5 (AES-256)	256	3329	4	(2, 2)	(11, 5)

2.4 *Kyber-PKE and Kyber-KEM*

Algorithms 1, 2, and 3 show the key generation, encryption, and decryption of Kyber-PKE, respectively. In these algorithms, extendable-output function (XOF) and pseudo-random function (PRF) mean an extendable output function and a pseudorandom function, respectively. The Kyber reference implementation uses secure hash algorithm and keccak (SHAKE)-128 and SHAKE-256 as an instantiation of XOF and PRF, respectively. Parse is a function used to convert randomly generated byte streams to NTT representations. CBD is a centered binomial distribution function. Encode and Decode are used for the serialization and deserialization of Kyber data, respectively. Compress and Decompress are used to reduce the size of a ciphertext. The details of these functions can be found in the specifications of Crystals-Kyber.

Algorithm 1 generates public and private key pairs (*pk*, *sk*). A public matrix A is generated based on the seed value ρ, and the elements in the matrix are converted into the NTT domain in step 2. The secret key vector s and error vector e are sampled through steps 4 to 5. Then, a public key vector is generated by computing $\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{A}} \circ \hat{\boldsymbol{s}} + \hat{\boldsymbol{e}}$ in step 8 (Note that the computation is conducted in the NTT domain for efficiency).

Algorithm 1: Kyber-PKE Key Generation

```
Ensure: secret key and public key pair (pk, sk)
1. $d \leftarrow B^{32}$ //generate 32-byte random seed
2. $(\rho, \sigma) \leftarrow G(d)$ //generate two seed values for matrix generation and noise generation
3. $\hat{A} \leftarrow \mathrm{Parse}(\mathrm{XOF}(\rho))$ //generate matrix in NTT domain with rejection sampling
4. $s \leftarrow \mathrm{CBD}_{\eta_1}(\mathrm{PRF}(\sigma, N))$ //generate secret key vector $\boldsymbol{s}$ with CBD sampling
5. $e \leftarrow \mathrm{CBD}_{\eta_1}(\mathrm{PRF}(\sigma, N))$ //generate noise vector $\boldsymbol{e}$ with CBD sampling
6. $\hat{s} \leftarrow \mathrm{NTT}(s)$ //convert secret key vector into NTT domain
7. $\hat{e} \leftarrow \mathrm{NTT}(e)$ //convert noise vector into NTT domain
8. $\hat{t} \leftarrow \hat{A} \circ \hat{s} + \hat{e}$ //compute matrix–vector multiplication with point-wise multiplication
9. $pk \leftarrow \mathrm{Encode}_{12}(\hat{t})||\rho$ //encode public key with seed for generating the matrix
10. $sk \leftarrow \mathrm{Encode}_{12}(\hat{s})$ //encode secret key
11. Return $(pk, sk)$
```

Algorithm 2: Kyber-PKE Encryption

```
Require: public key pk, message m, random coins $r \in B^{32}$
Ensure: ciphertext $c = (c_1, c_2)$
1. $\hat{t} \leftarrow Decode_{12}(pk)$ //decode and obtain public key and seed for matrix generation
2. $\rho \leftarrow pk$
3. $\hat{A} \leftarrow \mathrm{Parse}(\mathrm{XOF}(\rho))$ //generate matrix in NTT domain
4. $r \leftarrow \mathrm{CBD}_{\eta_1}(\mathrm{PRF}(r, N))$ //generate random vector with CBD sampling
5. $e_1 \leftarrow \mathrm{CBD}_{\eta_2}(\mathrm{PRF}(r, N))$ //generate random error vector with CBD sampling
6. $e_2 \leftarrow \mathrm{CBD}_{\eta_2}(\mathrm{PRF}(r, N))$ //generate random polynomial with CBD sampling
7. $\hat{r} \leftarrow \mathrm{NTT}(r)$ //convert random vector into NTT domain
8. $u \leftarrow \mathrm{NTT}^{-1}(\hat{A} \circ \hat{r}) + e_1$ //matrix and vector multiplication and then convert normal domain
9. $c_1 \leftarrow Encode_{d_u}(\mathrm{Compress}_q(u, d_u))$ //encode the first part of ciphertext
10. $v \leftarrow \mathrm{NTT}^{-1}(\hat{t}^T \circ \hat{r}) + \mathrm{Decompress}_q(Decode_1(m), 1)$ //embedding message into the second part of
    ciphertext
11. $c_2 \leftarrow \mathrm{Encode}_{d_v}(Compress_q(v, d_v))$ //encode the second part of ciphertext
12. Return $c = (c_1, c_2)$
```

Algorithm 3: Kyber-PKE Decryption

```
Require: secret key $sk$, ciphertext $c = (c_1,\ c_2)$
Ensure: message $m$
  1. $u \leftarrow Decompress_q(Decode_{d_u}(c_1,\ d_u)$ //decode and decompress the first part of ciphertext
  2. $v \leftarrow Decompress_q(Decode_{d_v}(c_2,\ d_v))$ //decode and decompress the second part of ciphertext
  3. $\hat{s} \leftarrow \mathrm{Decode}_{12}(sk)$ //decode the secret key vector
  4. $m \leftarrow Encode_1(Compress_q(v - \mathrm{NTT}^{-1}(\hat{s}^T \circ \mathrm{NTT}(u)),\ \ 1)$ //decrypt the ciphertext
  5. Return $m$
```

Finally, a public key pk and a secret key sk are generated using the encoding function in steps 9 and 10. Note that to reduce the size of pk, ρ is appended in pk rather than containing the whole **A**. In other words, if a recipient receives pk, he/she can compute **A** from the ρ value.

Algorithm 2 generates ciphertext c from a message m using the public key pk. The generated ciphertext consists of two parts (c_1, c_2). Through steps 1–3, the public key factors $\hat{t}$ and $\hat{A}$ are recovered from pk. A random vector **r**, a random error vector $\mathbf{e_1}$, and an error polynomial e_2 are sampled through steps 4–6. c_1, the first part of the ciphertext, is generated through steps 8 to 9. Note that matrix by vector multiplication is computed in the NTT domain in step 8. c_2, the second part of the ciphertext, is computed in steps 10 and 11. Vector by vector multiplication is also computed in the NTT domain in step 10.

Algorithm 3 recovers the message m from the ciphertext with the secret key sk. The original message m comes from c_2 by computing step 4. In other words, the randomness is removed by subtracting $\mathrm{NTT}^{-1}(\hat{s}^T \circ \mathrm{NTT}(u))$ from v.

As aforementioned, Kyber-KEM is constructed using Kyber-PKE, which consists of key generation, encapsulation, and decapsulation. For INC-CCA2 security, each algorithm of Kyber-KEM is different from the corresponding algorithm of Kyber-PKE. The most different part is between the decapsulation of Kyber-KEM and the decryption of Kyber-PKE. Namely, the decapsulation executes an additional process that encrypts the recovered message to check whether the generated ciphertext is the same as the received ciphertext. The detailed Kyber-KEM process can be found in the algorithm specification and supporting document [2].

2.5 WebAssembly and SIMD Programming

Since its release in 2017, Wasm has provided a portable execution framework for developing high-performance web applications [9,10]. Because Wasm enables C/C++-based native codes to run on web applications, Wasm modules with Javascript applications provide not only portability but also high performance in the web environment. Regarding the secure execution of Wasm-based modules, because Wasm provides a memory-safe, sandboxed execution environment inside JavaScript virtual machines, the used browser's security policies are applied to them. Furthermore, with Wasm, it is possible to process multiple data in parallel in an SIMD manner. Namely, SIMD instructions in Wasm process several data units packed in a 128-bit register in parallel. When the element size is 8-bit, sixteen elements can be packed in a register. Similarly, eight (resp. four) and 16-bit (resp. 32-bit) data can be packed in a register. However, currently, Wasm supports only 128-bit SIMD instructions, whereas AVX2 [16] in typical CPUs provides a set of 256-bit registers. *wasm_simd128.h* defines several SIMD intrinsic functions.

Fig. 1 shows the process of building a Wasm module and using it in a web application. First, native code modules developed in C/C++ are compiled with Emscripten SDK, producing a Wasm module (.wasm extension). The Wasm module is a binary bytecode and consists of several standard sections such as type, import, and function. When the module is loaded into a web browser supporting Wasm, it verifies whether the module is valid, and the virtual bytecode is compiled into a machine code of the device (x86 or ARM) the browser is running on.

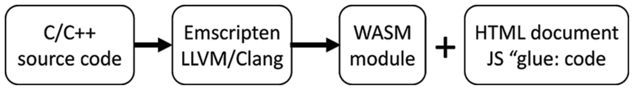

Figure 1: Wasm conversion process

Regarding crypto optimization using Wasm, in 2021, Bosun Park et al. presented a crypto library based on Wasm and Javascript for providing portable, efficient, and secure communication in Web-based IoT applications [11]. However, there is no research on optimizing the PQC algorithm using Wasm yet. Thus, we present an optimized Crystals-Kyber-KEM library based on Wasm for efficient and secure key establishment in the quantum computing era.

3 Related Works

Thus far, research on Kyber performance optimization has been focused on general-purpose CPUs (with AVX2 instructions) and ARM-based embedded processors (Cortex-M4, ARMv8-A series). However, there is little research on developing portable Lattice-based PQC libraries based on Javascript for secure web communication. Their goal has been to prove the feasibility of using Lattice-based cryptography in the web environment.

As a first step, in 2016, Yuan et al. presented a portable implementation of several Lattice-based cryptosystems (NTRU, NTRU-IEEE07, Regev's LWE, LPR10-LWE, LP10 Ring-LWE, and LP11) based on Javascript on several web browsers (Google Chrome, Firefox, Opera, and Internet Explorer; Android and Tessel) [6]. They optimized the core parts of Lattice-based cryptosystems such as polynomial multiplication and discrete Gaussian sampling and measured the performance of key generation, encryption, and decryption on their target web browsers. Furthermore, notably, they analyzed the proportion of sub-operations constituting each key generation, encryption, and decryption. However, their target algorithms do not belong to the NIST PQC Round 3 finalist (Parameters used by NTRU implementation in [6] differ from those of NTRU in NIST PQC Round 3 finalist).

In 2019, Ye Yuan et al. presented Javascript implementations of five Lattice-based cryptosystems (Lizard [17], Ring-Lizard [18], Kyber [2], FrodoKem [19], and NewHope [20]) for providing a portable key exchange protocol in the web environment. They measured their Javascript implementations on several web browsers and analyzed the proportion of sub-operations (matrix multiplication, error sampling, NTT-based polynomial multiplication, and so on). In their implementation, Kyber and NewHope outperformed Lizard, Ring-Lizard, and Frodo. Among their target algorithms, only Kyber belongs to the NIST PQC Round 3 finalist. However, the Kyber parameters used in their implementation are not the same as those presented in Kyber's final round specification.

In 2021, Anton Tutoveanu presented a Kyber implementation using the parameter presented in the NIST PQC Round 3 finalist based on Javascript and benchmarked the performance of the key establishment process using Kyber when a client received 150,000 bytes of encrypted data from a server [8]. They stated that the key establishment process using Kyber accounted for approximately 23% of the total transaction time.

Until now, several Lattice-based cryptosystems have been implemented using Javascript for portability. There is no research utilizing the latest Wasm technology. Thus, we use Wasm and take full advantage of SIMD instructions available with Wasm for not only portability but also high performance.

Regarding Lattice-based cryptosystems, several studies have been conducted for constructing cryptographic protocols or secure applications such as medical big data management systems [21], blind signature schemes for blockchain-enabled systems [22], and secure distributed healthcare service design [23].

4 Proposed Optimization Strategies of Crystals-Kyber

Our Kyber implementation is based on Wasm and takes full advantage of SIMD functionality in Wasm. First, we profile the performance of the latest Kyber Javascript implementation [8]. Our strategy is to identify the time-consuming parts of the Kyber Javascript implementation and then replace them with our proposed Wasm modules for both portability and high performance. In other words, our Kyber library is implemented with Javascript language for portability, and the time-consuming parts are implemented with Wasm modules for efficiency.

4.1 Performance Profiling

We have profiled each key generation, encryption, and decryption of the latest Kyber Javascript implementation. We have measured sub-operations constituting each key generation, encryption, and decryption using the performance.now() method, which can measure time by milliseconds. Fig. 2 shows the profiling results for Kyber key generation, encapsulation, and decapsulation. From the figure, the first two time-consuming parts are the generation of matrix **A** and hash functions (SHA-3-224, SHA-3-512, and SHAKE). Note that the matrix generation significantly uses SHA-3-related functions. Although the ratio of $\mathbf{A} \circ \mathbf{s}$, the multiplication between a matrix and a vector, seems to be smaller than that of matrix generation and hash functions, the main reason for this is that the codes used for SHA-3 and its variants (SHAKE-128 and SHAKE-256) in [8] are not optimized. Profiling results from the other literature [24] show that the overhead of $\mathbf{A} \circ \mathbf{s}$ is larger than that of matrix generation. Furthermore, several CPUs and ARM-based MCUs (Cortex-A53) support a hardware accelerator of SHA hash function (SHA-1 and SHA-256) with a dedicated instruction set. Although SHA-3 is not currently supported yet, it is expected that upcoming CPUs/MCUs will support an SHA-3 hardware accelerator. Thus, in our implementation, we aim at optimizing the performance of NTT-based polynomial multiplication, which is the core operation when computing matrix by vector multiplication and vector by vector multiplication with SIMD intrinsic functions available in Wasm. Algorithms 1, 2, and 3 show the execution of NTT conversions, point-wise multiplication, and inverse NTT conversion.

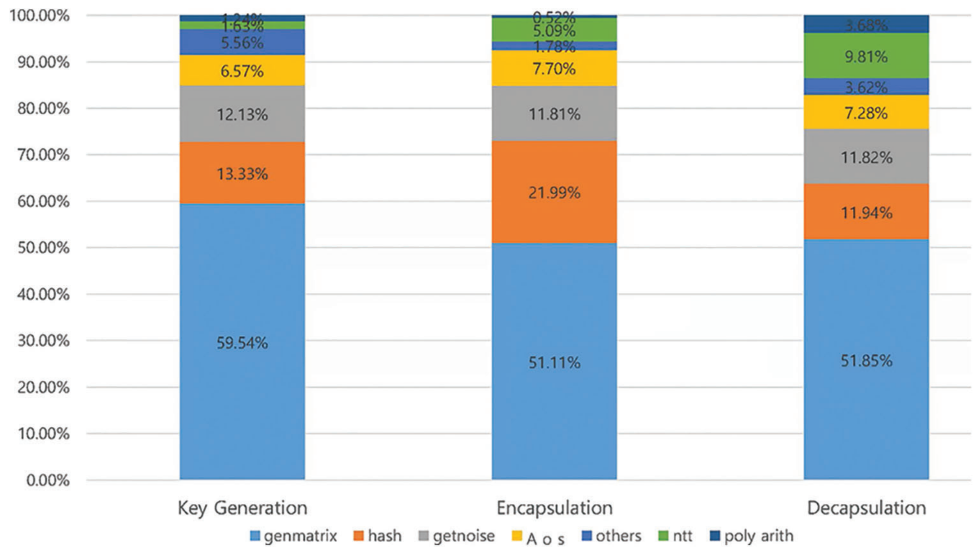

Figure 2: Timing profile of kyber operation

4.2 SIMD Intrinsics on Wasm

Table 2 introduces Wasm intrinsic functions used to implement our parallel NTT-based polynomial multiplication. Wasm SIMD Intrinsic functions provide 128-bit vector registers and related arithmetic functionalities similar to Intel's streaming SIMD extensions (SSE). However, Wasm SIMD is under development, so its support for SIMD is incomplete. Thus, a Wasm SIMD library to directly convert crypto software implemented with SSE extension intrinsic to a module using Wasm SIMD intrinsics is currently unavailable. Refer to [10] for the current limitations of Wasm SIMD.

Table 2: SIMD intrinsic functions on wasm

Intrinsics	Description
$v \leftarrow$ wasm_v128_load(*addr*)	Load 16-byte data from the *addr* memory address to vector v
wasm_v128_store(*addr*, x)	Store 16-byte data in vector x to the *addr* memory location
$v \leftarrow$ wasm_i16x8_splat(x)	Construct a vector v with 16-bit x replicated to 8 lanes
$v \leftarrow$ wasm_i32x4_extmul_low_i16x8 (x, y)	Multiply low parts of 16-bit lane-wise integer extended multiplication (namely, it multiplies lower 16-bit four lanes)
$v \leftarrow$ wasm_i32x4_extmul_high_i16x8 (x, y)	Multiply high parts of 16-bit lane-wise integer extended multiplication (namely, it multiplies higher 16-bit four lanes)
$v \leftarrow$ wasm_i16x8_mul(x, y)	Computes 16-bit lane-wise wrapping integer multiplication (namely, higher 16-bit of each 32-bit multiplication results is discarded)
$v \leftarrow$ wasm_i16x8_shuffle(x, y, *imm*)	Returns a new vector with 16-bit lanes selected from the lanes of the two input vectors x and y specified in the 16-byte wide immediate mode operand *imm*
$v \leftarrow$ wasm_i32x4_add(x, y)	Compute 32-bit lane-wise wrapping integer addition
$v \leftarrow$ wasm_i32x4_shr(x, *scalar*)	Shift the bits in each lane to the right by *scalar* bits

4.3 Optimization of NTT-Based Polynomial Multiplication with Wasm SIMD

NTT-based polynomial multiplication consists of three steps: NTT conversion, component-wise (point-wise) multiplication, and inverse NTT conversion. NTT conversion step converts a polynomial a over $R_q = Z_q[X]/(X^{256} + 1)$ to 128 sub-polynomials over $X^2 - \zeta$. Two polynomials a and b over R_q need to be converted into the NTT domain. Point-wise multiplication computes multiplications between 128 sub-polynomials constituting a and 128 sub-polynomials constituting b. Finally, the sub-polynomials from point-wise multiplications are converted to a polynomial c over R_q. Because each coefficient in Kyber is stored in a 16-bit integer value, eight coefficients can be loaded in a 128-bit vector. Thus, our parallel version of each NTT conversion, point-wise multiplication, and inverse NTT conversion processes eight coefficients in parallel. In the following subsections, we describe the target codes (NTT conversion, point-wise multiplication, and inverse NTT conversion) and how to convert them to parallel codes with Wasm intrinsic functions.

4.3.1 Proposed Parallel NTT Conversion

Listing 1 shows the original `ntt` function that converts a polynomial over R_q to the NTT domain. Because a polynomial contains 256 coefficients, and each coefficient is less than `q (=3329),` `int16_t r[ 256]` is used to express a polynomial. This `ntt` function is an in-place version, which means that the intermediate results are updated to the input array `r`. In the `ntt` function, the zeta values are precomputed and stored in the global memory area (`zetas` is an array containing twiddle factor values over $R_q = Z_q[X]/(X^{256} + 1)$. Steps 12–14 are called Butterfly operations because the symmetric computations are related such as `r[ j + len] = r[ j] − t` and `r[ j] = r[ j] + t`. First, `r[ j + len]` is multiplied by a twiddle factor zeta. Because two 16-bit words are multiplied, the result needs to be reduced over q. This reduction is conducted with Montgomery-based reduction. Note that `fqmul(x, y)` in the function multiplies `x` and `y` and then reduces the result of multiplication with the Montgomery reduction described in Listing 2. Then, `t`, the result of the reduction, is either added to or subtracted from `r[ j]`.

Listing 1: ntt Function for NTT domain conversion

```
void ntt(int16_t r[256]) {
  unsigned int len, start, j, k;
  int16_t t, zeta;

  k = 1;

  for(len = 128; len >= 2; len >>= 1) {
    for(start = 0; start < 256; start = j + len) {
      zeta = zetas[k++];
      for(j = start; j < start + len; j++) {

        t = fqmul(zeta, r[j + len]);
        r[j + len] = r[j] - t;
        r[j] = r[j] + t;
      }
    }
  }
}
```

We parallelize the internal computation of `ntt` with Wasm SIMD intrinsic functions (Wasm SIMD supports only 128-bit wide vector registers). Because each coefficient in a polynomial is less than q, which is approximately 12-bit, eight coefficients can be loaded in a vector register of Wasm SIMD. Namely, we process eight execution of for-loop simultaneously with Wasm SIMD. For efficiency, we unroll the nested for-loops in `ntt` for easy application of the SIMD technique.

Fig. 3 shows the overall structure of the proposed parallel NTT conversion process. For simplicity, we use a polynomial with 32 coefficients rather than 256 coefficients. First, eight coefficients are loaded into a vector with the wasm_v128_load() function. Note that the related twiddle factors are also loaded into a vector. Then, vector by vector multiplication is performed using two intrinsic functions: wasm_i32 × 4_extmul_low_i16 × 8() and wasm_i32 × 4_extmul_high_i16 × 8(). In other words, wasm_i32 × 4_extmul_low_i16 × 8() multiplies the lower 16-bit four lanes in the blue-colored vector and lower 16-bit four lanes in the green-colored vector, and the 32-bit four results are stored in the yellow-colored vector (low parts). Higher part multiplications are executed in the same way using wasm_i32 × 4_extmul_high_i16 × 8(), and the results are stored in the yellow-colored vector (high parts). Because each of the eight 32-bit results held in the two vectors is larger than q, it needs to be reduced. For efficient reduction, Montgomery reduction is applied.

Listing 2 shows the reduction algorithms used for NTT-based polynomial multiplication. montgomery_reduce() reduces 32-bit data to a value less than q with simple arithmetics (two-word multiplications, one right shift, and one subtraction). Note that QINV and KYBER_Q are predefined values for the Montgomery reduction. In our parallel implementation, eight 32-bit results are reduced with our parallel implementation of the montgomery_reduce() function. It uses wasm_i16 × 8_mul() for eight computations of a ∗ QINV in step 5 and uses wasm_i32 × 4_extmul_low_i16 × 8() and wasm_i32 × 4_extmul_high_i16 × 8() for eight computations of t ∗ KYBER_Q in step 6. The subtraction and right shift in step 6 are executed with wasm_i32 × 4_sub() and wasm_i32 × 4_shr(). Finally, each of the reduced results held in the two vector registers is either added or subtracted with r[j] and then stored in r[j] and `r[ j + len]` , respectively. As the result of NTT conversion, a 255-th degree polynomial over R_q is converted into 128 1-th degree sub-polynomials.

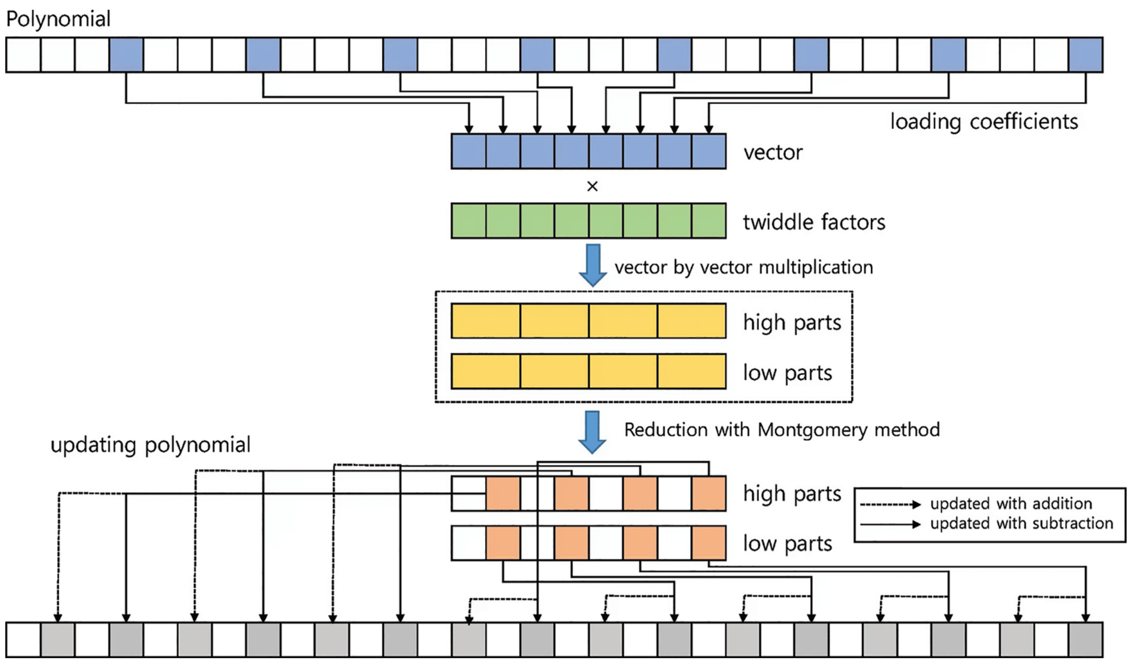

Figure 3: The structure of proposed parallel NTT conversion

Listing 2: ntt function for NTT domain conversion

```
int16_t montgomery_reduce(int32_t a)
{
  int16_t t;

  t = (int16_t)a*QINV;
  t = (a - (int32_t)t*KYBER_Q) >> 16;
  return t;
}

int16_t barrett_reduce(int16_t a)
{
    int16_t t;
    const int16_t v = ((1<<26) + KYBER_Q/2)/KYBER_Q;

    t  = ((int32_t)v*a + (1<<25)) >> 26; t *= KYBER_Q;
    return a - t;
}
```

4.3.2 Proposed Parallel Point-Wise Multiplication

Listing 3 shows the original poly_basemul_montgomery function that multiplies two polynomials (a and b) in the NTT domain in a point-wise multiplication manner. Each of a and b consists of 128 1-th degree sub-polynomials such as $a = (a_{255}X + a_{254}, \ldots, a_3X + a_2, a_1X + a_0)$ and $b = (b_{255}X + b_{254}, \ldots, b_3X + b_2, b_1X + b_0)$. Point-wise multiplication computes $c = (c_{255}X + c_{254}, \ldots, c_3X + c_2, c_1X + c_0)$, where $(c_{2i+1}X + c_{2i}) = (a_{2i+1}X + a_{2i}) \circ (b_{2i+1}X + b_{2i})$. Note that one point-wise multiplication calls five fqmul functions because $c_{2i+1}X = (a_1 \cdot b_0 + a_0 \cdot b_1)X$, $c_{2i} = (a_0 \cdot b_0 + a_1 \cdot b_1 \cdot \zeta^{2i+1})$, where $X^2 = \zeta^{2i+1}$.

Listing 3: poly_basemul_montgomery function for point-wise multiplication

```
void poly_basemul_montgomery(poly *r, const poly * a, const poly *b)
{
  unsigned int i;
  for(i=0;i<KYBER_N/4;i++) {
     basemul(&r->coeffs[4*i], &a->coeffs[4*i], &b-> coeffs[4*i],
     zetas[64+i]);
     basemul(&r->coeffs[4*i+2], &a->coeffs[4*i+2], &b->coeffs[4*i+2],
     -zetas[64+i]);
  }
}

void basemul(int16_t r[2], const int16_t a[2], const int16_t b[2],
int16_t zeta)
{
    r[0]  = fqmul(a[1], b[1]);
    r[0]  = fqmul(r[0], zeta);
    r[0] += fqmul(a[0], b[0]);
    r[1]  = fqmul(a[0], b[1]);
    r[1] += fqmul(a[1], b[0]);
}
```

Fig. 4 shows the overall structure of the proposed parallel point-wise multiplication process. For simplicity, we use a polynomial with 32 coefficients rather than 256 coefficients. The proposed method computes eight point-wise multiplications in parallel. First, eight sub-polynomials of a (resp. b) are loaded into two vector registers v_1 and v_0 (resp. v_3 and v_2). Furthermore, coefficients with (resp. without) X are loaded into v_3 and v_1 (resp. v_2 and v_0) vector registers. Twiddle factors are also loaded into a temporary vector register v_t. Then, v_1, v_3, and v_t are multiplied with two fqmul executions and stored in a temporary register v_{lower}. v_0 and v_2 are also multiplied with one fqmul execution, and the result is accumulated to v_{lower} register. The coefficients in v_{lower} are (e.g., $(c_{14}, c_{12}, \ldots, c_2, c_0)$) without X. The accumulation of two vector multiplication results (e.g., (v_1 and v_2, and v_0 and v_3)) with two fqmul executions constructs coefficient c_{2i+1} (e.g., $c_{15}, c_{13}, \ldots, c_3, c_1$)) with X. The main computation in the proposed parallel point-wise multiplication is fqmul. In our parallel fqmul implementation, first, eight coefficients of two vectors are multiplied with wasm_i32 × 4_extmul_low_i16 × 8() and wasm_i32 × 4_extmul_high_i16 × 8(). Then, the results of multiplications are reduced with Montgomery reduction, which is described in the proposed parallel NTT conversion.

4.3.3 Proposed Parallel Inverse NTT Conversion

Listing 4 shows the original `invntt` function, which converts a polynomial (consisting of 128 1-th degree sub-polynomials) in the NTT domain to a polynomial over R_q. The structure of `invntt` is similar to that of `ntt`, except for its sequence of operations. The execution of addition and subtraction precedes multiplication with fqmul (the zeta value used in the multiplication is the inverse value of zeta in the `ntt` function). Because the result of addition may become larger than q, it needs to be reduced using the Barrett reduction method. The function for Barrett reduction is given in Listing 2, and it is related to two-word multiplications, one addition, one subtraction, and one right shift (v value is the precomputed value (1 << 26) and is determined at compile time). The result of Barrett reduction and the result of fqmul are stored in `r[j]` and `r[j+len]`, respectively. Note that the final 256 coefficients (steps 17–21) need to be multiplied by f value (2^{-7} mod q) with fqmul because coefficients are doubled at each layer as the result of addition and subtraction.

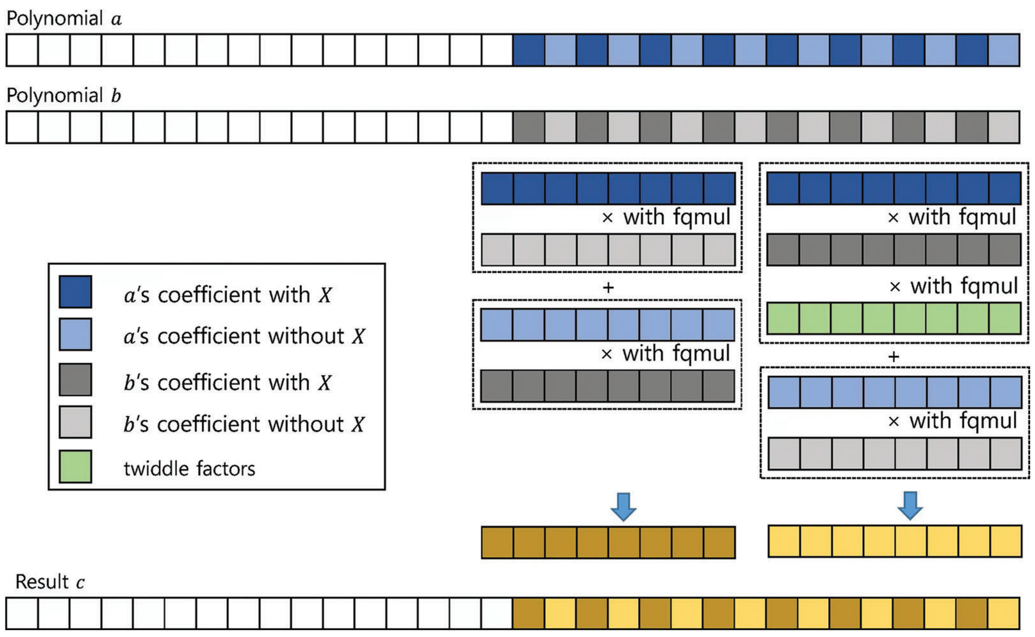

Figure 4: The structure of proposed parallel point-wise multiplication

Listing 4: invntt function for inverse NTT conversion

```
1  void invntt(int16_t r[256]) {
2     unsigned int start, len, j, k; int16_t t, zeta;
3     const int16_t f = 1441; // mont^2/128
4
5     k = 127;
6     for(len = 2; len <= 128; len <<= 1) {
7         for(start = 0; start < 256; start = j + len) {
8            zeta = zetas[k--];
9              for(j = start; j < start + len; j++) {
10                 t = r[j];
11                 r[j] = barrett_reduce(t+r[j+len]);
12                 r[j+len] = r[j+len]-t;
13                 r[j+len] = fqmul(zeta, r[j+len]);
14             }
15         }
16    }
17    for(j = 0; j < 256; j++)
18    {
19        R[j] = fqmul(r[j], f);
20     }
21 }
```

Fig. 5 shows the overall structure of the proposed parallel inverse NTT conversion process. For simplicity, we use a polynomial with 32 coefficients rather than 256 coefficients. Eight `r[ j]` values and eight `r[ j + len]` values are loaded into two vectors (vector1 and vector2), respectively. Then, vector1 and vector2 are added using wasm_i16 × 8_add(), and the results are reduced by our parallel Barrett reduction method. Our parallel implementation of Barrett reduction uses wasm_i32 × 4_extmul_low_i16 × 8() and wasm_i32 × 4_extmul_high_i16 × 8() to compute eight multiplications of $v*a$. Eight multiplications of $t * =$ KYBER_Q are computed with wasm_i16 × 8_mul_low_i16 × 8(). The results of Barrett reduction are stored in their original memory locations `r[ j]` . The results of (vector1–vector2) are multiplied with twiddle factors and then reduced by Montgomery reduction. The final results of the reduction are stored in their memory addresses `r[ j + len]`.

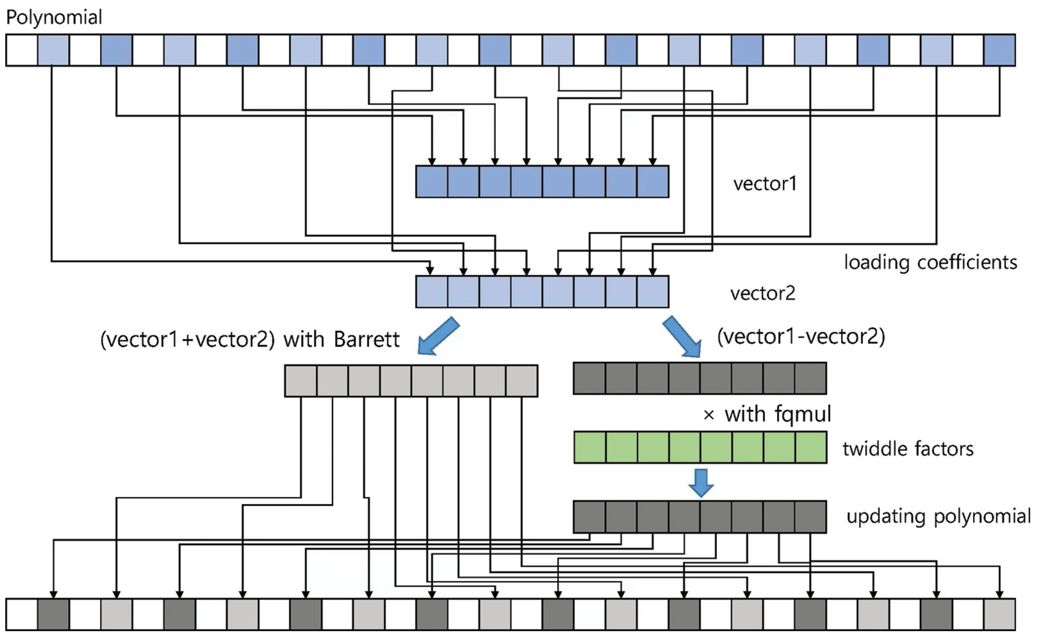

Figure 5: The structure of proposed parallel inverse NTT conversion

5 Performance Analysis

In this section, we analyze our proposed implementations and compare them with those of previous studies. For efficiency comparison, we measure the execution times and compare them with those of previous studies. Because our software is a web-based cryptographic library, it will run on web browsers on personal computers or smartphones. Thus, the code size is not a meaningful comparison factor. For performance analysis and comparison, we use three browsers: Google Chrome, Firefox, and Microsoft Edge. Table 3 shows the experimental environment for performance analysis.

First, we analyze the performance of the proposed parallel implementation of NTT-based polynomial multiplication and compare it with the previous implementations. Table 4 compares the performance of the proposed implementation with the latest Crystals-Kyber Javascript implementation from [8]. In Table 4, Wasm means NTT-based polynomial multiplication steps implemented with only the *C* language using the Wasm framework. This Works means that each step in NTT-based multiplication is implemented with Wasm SIMD functionality. From the table, the proposed SIMD implementation outperforms the latest Javascript implementation from [8] by ×6.83, ×4.95, and ×7.99 in terms of NTT conversion, point-wise multiplication, and inverse NTT conversion. Compared with our naive Wasm implementation, our SIMD implementation provides ×2.24, ×2.344, and ×2.113 superior performance in terms of NTT conversion, point-wise multiplication, and inverse NTT conversion. Although we parallelize the internal process of NTT-based multiplication by eight data units, the performance improvement is less than eight times. This is because the overhead of data conversion between the Javascript and Wasm data types is large, as well as loading data to SIMD registers and storing them in memory, is significant.

Table 3: Performance measurement environment

Operating system	Windows 10 Pro 64-bit/Ubuntu Linux 16.04
CPU/RAM	Intel i9-10900 K 3.70 GHz/32 GB
SW	Chrome
	Firefox
	Microsoft Edge
Language	Javascript
	WebAssembly

Table 4: Performance of NTT-based polynomial multiplication (Measured by milliseconds. The performance improvement ratio was obtained by comparison with JS [8].)

Methods	NTT	Point-wise multiplication	InvNTT
JS [8]	0.01914	0.0262	0.02717
Wasm	0.0065	0.012	0.0082
This Work	0.0028	0.0053	0.0034
	(×6.83)	(×4.95)	(×7.99)

We integrate the proposed parallel implementation of NTT-based polynomial multiplication into Crystals-Kyber software. Tables 5–7 compare our Kyber software with the latest Kyber Javascript implementation from [8] on Google Chrome, Firefox, and Microsoft Edge in terms of key generation, encapsulation, and decapsulation. For key generation (resp. encapsulation and decapsulation), our software provides ×4.02 (resp. ×3.42 and ×3.41), ×4.32 (resp. ×3.52 and ×3.44), and ×4.1 (resp. ×3.44 and ×3.38) improved performance on the three browsers, respectively. The proposed Kyber implementation based on Wasm SIMD outperformed the previous work.

Table 5: Performance measurement results with google chrome (Measured by milliseconds)

Methods	KeyGen	Encapsulation	Decapsulation
JS [8]	1.975	2.3319	2.2709
This Work	0.491	0.6817	0.6661
	(×4.02)	(×3.42)	(×3.41)

Table 6: Performance measurement results with firefox (Measured by milliseconds)

Methods	KeyGen	Encapsulation	Decapsulation
JS [8]	4.7037	5.9620	6.1148
This Work	1.0888	1.6938	1.7776
	(×4.32)	(×3.52)	(×3.44)

Table 7: Performance measurement results with microsoft edge (Measured by milliseconds)

Methods	KeyGen	Encapsulation	Decapsulation
JS [8]	1.9639	2.3402	2.2163
This Work	0.479	0.6803	0.6557
	(×4.1)	(×3.44)	(×3.38)

6 Conclusion

In this study, we present the first implementation of Crystals-Kyber with Wasm in the web environment. Using the SIMD functionality of the Wasm framework, we parallelize NTT-based polynomial multiplication with intrinsic functions. Our optimized parallel implementation of NTT-based polynomial multiplication outperforms the previous Javascript NTT implementation by ×6.83, ×4.95, and ×7.99 in terms of NTT conversion, point-wise multiplication, and inverse NTT conversion, respectively. Based on the proposed parallel NTT-based multiplication, we propose a Crystals-Kyber Wasm module for providing security levels 1, 3, and 5. The proposed Crystals-Kyber module can be used by various web applications for key establishment processes in TLS v1.3, SSH (Secure Shell), IPSec IKE (Internet Key Exchange), and so on. Our future work is to design and implement a hybrid key exchange mode using Crystals-Kyber and classical ECDH, which are described in NIST 800-56C rev2 [25].

Funding Statement: This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-01019, Development of eSIM security platform technology for edge devices to expand the eSIM ecosystem). This was partly supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2022-RS-2022-00164800) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] NIST, "Nist post-quantum cryptography," 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography.

[2] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz *et al.,* "Crystals-kyber," 2021. [Online]. Available: https://pq-crystals.org/kyber/index.shtml.

[3] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe *et al.,* "Crystals-dillithium," 2021. [Online]. Available: https://pq-crystals.org/dilithium/index.shtml.

[4] T. Prest, P. -A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky *et al.,* "Falcon," 2021. [Online]. Available: https://falcon-sign.info.

[5] J. -P. Aumasson, D. J. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder *et al.,* "Sphincs+," 2021. [Online]. Available: https://sphincs.org/index.html.

[6] Y. Yuan, C. -M. Cheng, S. Kiyomoto, Y. Miyake and T. Takagi, "Portable implementation of lattice-based cryptography using javascript," *International Journal of Networking and Computing*, vol. 6, no. 2, pp. 309–327, 2016.

[7] Y. Yuan, J. Xiao, K. Fukushima, S. Kiyomoto and T. Takagi, "Portable implementation of lattice-based cryptography using javascript," *Security and Communication Networks*, vol. 2018, pp. 9846168:1–9846168:14, 2018.

[8] A. Tutoveanu, "Active implementation of end-to-end post-quantum encryption," *Cryptology ePrint Archive*, Report 2021/356, 2021. [Online]. Available: https://eprint.iacr.org/2021/356.

[9] WebAssembly Community Group, "Webassembly," 2022. [Online]. Available: https://webassembly.org.

[10] WebAssembly Community Group, "Porting simd code targeting webassembly," 2022. [Online]. Available: https://emscripten.org/docs/porting/simd.html.

[11] J. S. BoSun Park and S. C. Seo, "Efficient implementation of a crypto library using web assembly," *MDPI Electronics*, vol. 9, no. 11–1839, pp. 1–23, 2020.

[12] C. Agarwal and R. Burrus, "Fast convolution using fermat number transforms with applications to digital filtering," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 2, no. 22, pp. 87–97, 1974.

[13] G. Seiler, "Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography," *IACR Cryptol. ePrint Arch*, 2021. [Online]. Available: https://eprint.iacr.org/2021/356.

[14] T. J. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematic of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[15] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit," in *Proc. AFIPS'66*, New York, NY, USA: Association for Computing Machinery, pp. 563–578, 1966.

[16] Intel, "Intel haswell new instruction descriptions," 2012. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2.html.

[17] J. H. Cheon, D. H. Kim and Y. Song, "Lizard: Cut off the tail! practical post-quantum public-key encryption from lwe and lwr," *IACR Cryptol. ePrint Arch*, 2016. [Online]. Available: https://eprint.iacr.org/2016/1126.pdf.

[18] J. H. Lee, D. Kim, H. Lee, Y. Lee and J. H. Cheon, "RLizard: Post-quantum key encapsulation mechanism for IoT devices," *IEEE ACCESS*, vol. 7, pp. 2080–2091, 2018.

[19] E. Alkim, J. W. Bos, L. Ducas, P. Longa, I. Mironov *et al.,* "Frodokem," 2021. [Online]. Available: https://frodokem.org.

[20] E. Alkim, R. Avanzi, J. W. Bos, L. Ducas, A. d. l. Piedra *et al.,* "Newhope," 2021. [Online]. Available: https://newhopecrypto.org.

[21] C. Li, M. Dong, J. Li, G. Xu, X. -B. Chen *et al.,* "Efficient medical big data management with keyword-searchable encryption in healthchain," *IEEE Systems Journal*, vol. 16, no. 4. pp. 1–12, 2022.

[22] C. Li, Y. Tian, X. Chen, and J. Li, "An efficient anti-quantum lattice-based blind signature for blockchain-enabled systems," *Information Sciences*, vol. 546, no. 6, pp. 253–264, 2021.

[23] C. Li, M. Dong, J. Li, G. Xu, X. Chen *et al.,* "Healthchain: Secure EMRs management and trading in distributed healthcare service system," *IEEE Internet of Things Journal*, vol. 8, no. 9, pp. 7192–7202, 2020.

[24] P. Sanal, E. Karagoz, H. J. Seo, R. Azarderakhsh and M. Mozaffari-Kermani, "Kyber on arm64: Compact implementations of kyber on 64-bit arm cortex-A processors," *IACR Cryptol. ePrint Arch*, 2021. [Online]. Available: https://eprint.iacr.org/2021/561.

[25] NIST, "Recommendation for key-derivation methods in key establishment schemes," 2020. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-56c/rev-2/final.