

RESEARCH ARTICLE

Vectorized Implementation of Kyber and Dilithium on 32-bit Cortex-A Series

YOUNGBEOM KIM^{ID1}, (Student Member, IEEE), SEUNGYONG YOON^{ID2}, (Member, IEEE),
AND SEOG CHUNG SEO^{ID1}, (Member, IEEE)

¹Department of Financial Information Security, Kookmin University, Seoul 02707, South Korea

²Cyber Security Research Division, Electronics and Telecommunications Research Institute, Daejeon 34129, Republic of Korea

Corresponding author: Seog Chung Seo (scseo@kookmin.ac.kr)

ABSTRACT In the field of Post-Quantum Cryptography (PQC), which typically demands more memory and relatively lower performance compared to Elliptic-Curve Cryptography (ECC), recent studies have been actively focused on neon-based parallel implementations for the 64-bit ARMv8-based Cortex-A series. However, research into implementing PQC on the widely adopted 32-bit ARMv7-based Cortex-A series remains insufficient. In this paper, we present the first instance of optimized implementation of Crystals-Kyber and Crystals-Dilithium, a Key Encapsulation Mechanism (KEM) and a Digital Signature Algorithm (DSA) selected by National Institute of Standards and Technology (NIST) for standardization, on a 32-bit ARMv7-based Cortex-A device. For computational efficiency, we finely tune widely used signed Montgomery multiplication and Barrett multiplication methods to take full advantage of the computational capabilities of NEON engine, a kind of Single-Instruction-Multiple-Data (SIMD) extension, available on the target device. Particularly, we propose improvements to internal parameters and operational techniques in Montgomery and Barrett arithmetic to preserve parallel processing logic. Moreover, we present an optimized merging technique tailored for the NEON engine of ARMv7, aimed at accelerating Number Theoretic Transform (NTT)-based polynomial multiplication. Compared to the state-of-the-art codes of PQM4, our approach achieves significant performance enhancements in Kyber and Dilithium: 62% (54%) for NTT, 50% (62%) for Point multiplication, and 56% (55%) for inverse NTT (NTT^{-1}). Regarding the complete schemes, our implementations outperform the vectorized reference implementations, showing improvements of 50% (14%) in Key Generation, 43% (41%) in Encapsulation (Signing), and 52% (21%) in Decapsulation (Verifying) processes for Kyber768 (Dilithium3), respectively.

INDEX TERMS ARMv7 Cortex-A series, NEON, efficient implementation, software optimization, vectorization implementation, embedded security, post-quantum cryptography, Crystals-Kyber, Crystals-Dilithium.

I. INTRODUCTION

The development of quantum computers is threatening the security of currently used public-key cryptosystems such as RSA, DSA, and ECC. Since 2016, the National Institute of Standards and Technology (NIST) has begun a competition for standardization of Post-Quantum Cryptography (PQC) and four algorithms (Crystals-Kyber [3], Crystals-Dilithium [4], Falcon [5], and SPHINCS+ [6]) were selected last year. Among four algorithms, Crystals-Kyber

(Kyber) is the only Key Encapsulation Mechanism (KEM) for Key Establishment, while other algorithms are Digital Signature Algorithm (DSA) for data authentication and non-repudiation. There is important consideration before the selected algorithms are standardized and used in actual fields. First, it is necessary to examine the current cryptographic libraries used in industrialization and applications. Famous cryptographic libraries include OpenSSL [7] and OpenSSH [8], with specialized libraries for embedded devices like mbedTLS [9] and AVRCrypto [10]. Most cryptography algorithms in these libraries are optimized for the purpose of each library. For instance, OpenSSL has several

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

build options considering the x86/x64 and AVX2 environments, and the code generated in this context reflects the results of implementation optimization. Similarly, mbedTLS and AVRCrypto have optimized codes for embedded environments. These codes are written in sophisticated assembly and are updated with the latest optimization techniques. The implementation examples seen in commercial libraries aim for light, fast, and safe operations. In other words, as the 5G industry evolves and applications become more complex, it is crucial for cryptographic systems designed for personal data protection to operate with minimal resources, including speed and memory. Especially now, with the growing interest in the Internet of Things (IoT), there have been countless implementation results for public-key schemes (ECC and Discrete Logarithm Problem) in embedded devices and high-end processors. These implementations can be found in the following survey paper [11].

Therefore, it's important to identify the current position of PQC implementation research before standardization progresses. After the PQC competition activated by NIST, most research focused on optimized implementation for the ARM Cortex-M4 [2], [12], [13], [14], performance evaluation device [15]. These latest optimized implementations for Cortex-M4 have been integrated into a performance benchmarking library named PQM4 [16]. Additionally, the PQClean [17] library exists, which seeks to eliminate dependencies on external libraries and pursue clean reference code. PQClean primarily reflects updates in reference code rather than implementation optimization. Since the final stages of the PQC contest, several optimization researches on various operational environments other than Cortex-M4 have been conducted. Notably, optimization results in diverse environments like AVR [18], Cortex-M3 [19], ARMv8-based Cortex-A series [20], [21], [22], AVX2 [23], and GPUs [24], [25] showcase what PQC might look like when incorporated into industrialization. In embedded environments like 8-bit AVR and Cortex-M3/M4, Kyber and Dilithium mainly progressed with a focus on memory optimization [26], [27] and implementation of modular arithmetic [1], [2], [12]. This introduces a perspective of memory optimization for embedded environments, similar to the approach of mbedTLS and AVRCrypto, not just speed. In high-performance computing environments like AVX2 and GPUs, where ample memory space exists, research proceeded with a focus on optimization for extreme performance enhancement.

Kyber and Dilithium specification makes use of Number-Theoretic Transform (NTT)-based method for efficient polynomial multiplication and NTT-based polynomial multiplication consists of three steps: NTT conversion, Point multiplication, and inverse NTT (NTT^{-1}) conversion. The main computations of these three steps are modular multiplication and modular addition/subtraction over $\mathbb{Z}_q$, for modulus q . For efficient constant-time modular reduction, Montgomery method and Barrett method have been widely used.

The current direction of optimized implementation for Kyber and Dilithium in the ARMv8-based Cortex-A series primarily focuses on maximizing the parallelism of NTT-based polynomial multiplication using NEON instructions and minimizing the cost incurred in modular multiplication. Considering that the Cortex-A series consists of processors, this strategy proposes that maximizing parallelism is more effective than optimizing memory efficiency. Recent research on Kyber [21], Dilithium [20], [28], and Falcon [22] in the ARMv8-based Cortex-A series reflects this perspective well.

ARMv7-Cortex-A-based Sensor nodes, automotive applications, home appliances, and medical devices are currently in active use within the industry [29]. Therefore, optimal implementation is essential to achieve both service quality and quantum security in these areas. Although the PQM4 library exists for Cortex-M4, which uses the ARMv7-ME core, we point out that PQM4 may not be the best choice for ARMv7-based Cortex-A series, as they can utilize the NEON engine and have ample memory space being processors. Furthermore, considering the active research in maximizing parallelism in AVX2 and ARMv8-based Cortex-A series, exploring the potential for PQC optimized implementation in ARMv7-based Cortex-A series is meaningful.

Therefore, this paper aims to achieve the first quantum security in the ARMv7-based Cortex-A series. We closely examine the existing approaches to optimize modular arithmetic and propose optimization strategies for the ARMv7-based Cortex-A series. The instruction set of ARMv7 (AArch32) differs from that of ARMv8 (AArch64), requiring a careful approach, and we also consider optimization strategies proposed for AVX2. Additionally, we analyze the modular multiplication method used in PQM4 in detail, ultimately proposing the best vectorized NTT-based polynomial multiplication approach for the ARMv7-based Cortex-A series.

We aim to maximize the utilization of the ARMv7 architecture's features. The instruction set applied to the Cortex-A series starts with ARMv7-A (AArch32) [30], and in 2013, ARMv8-A (AArch64) [31], the successor to ARMv7-A, was released. The most significant feature of ARMv8-A is the introduction of two sub-instruction sets, AArch32 and AArch64, allowing the use of a 64-bit architecture through AArch64. Since then, most new Cortex-A series and custom architectures have been actively developed based on ARMv8-A. Notably, the recently released Apple MacBook series uses M1 and M2 chips designed based on the ARMv8 Cortex-A series. Like the AVX2 instruction set, the Cortex-A series includes a NEON engine capable of internal parallel operations. Unlike ARMv8-based device, ARMv7-based NEON Technology does not support multiplication for different word sizes, and even if expanded to 32-bit multiplication, there is no instruction to return the upper result. To achieve parallelization using only the limited instruction set of the ARMv7-A series, we propose 16/32-bit

modular arithmetic. We believe that our insights into modular arithmetic can be applied not only to Kyber and Dilithium but also to other algorithms requiring modular arithmetic, such as Saber, NTRU, and Falcon.

A. CONTRIBUTION

Our contributions in this work can be summarized as below:

1) *Presenting first PQC implementation on ARMv7-based Cortex-A series*

To our knowledge, this article represents the first PQC software operating on the ARMv7-based Cortex-A series. Particularly, our target algorithms, Crystals-Kyber and Crystals-Dilithium, are the winning algorithms of the NIST PQC competition and are candidates for future standardization. We propose a common optimized implementation for the core operation of these two Crystals algorithms, NTT-based polynomial multiplication, thereby providing the first quantum-secure result on the 32-bit Cortex-A series. Moreover, by presenting optimized implementation results for all security levels, we demonstrate that there are better alternatives on the Cortex-A series compared to the PQM4 implementations [1], [2] on Cortex-M4, which use the same ARMv7 Core, and compared to the reference implementations [3], [4]. Our implementation of Kyber512 (Kyber768 and Kyber1024) shows a performance improvement of 51% (50% and 47%), 48% (43% and 41%), and 57 (52% and 48%)% in Key Generation, Encapsulation, and Decapsulation process, respectively, compared to the vectorized compiling option set in the reference implementation [3]. Similarly, our implementation of Dilithium2 (Dilithium3 and Dilithium5) achieves performance improvements of 17% (14% and 16%)%, 41% (41% and 41%)%, and 24% (21% and 20%)% in Key Generation, Signing, and Verifying process, respectively, when compared to the vectorized compiling option set in the reference implementation [4].

2) *Presenting an implementation of Montgomery and Barrett Arithmetic tailored for ARMv7-based Cortex-A series*

After a thorough investigation of existing modular arithmetic, we analyze its suitability for the ARMv7-based Cortex-A series based on the NEON instruction set. Following this, we select Montgomery multiplication and Barrett reduction that are fully compatible with the instruction set for implementing Kyber and Dilithium. We revisit Montgomery multiplication to maintain maximal parallelism in vectorized operations, and reconfigure the parameter set for Barrett reduction, adjusting each operation to align with the NEON instruction set. Particularly, by carefully calculating the output range of the Montgomery and Barrett reduction when modifying parameters, our method not only offers flexibility applicable to both Kyber and Dilithium, but

also avoids introducing additional reductions in the overall Kyber and Dilithium implementation.

3) *Presenting a methodology for NTT implementation that incorporates the latest optimization techniques*

We incorporate all the latest optimization techniques proposed for NTT-based polynomial multiplication implementation and readjust them for the 32-bit Cortex-A series. First, we apply signed modular arithmetic and re-adjust the elegant modular multiplication method for parallel processing, as proposed in [21] and [23], to suit the NEON environment. Additionally, we revisit the application of the merging technique, extensively researched in [2] and [13], adapting it to the context of our paper. Moreover, by pre-computing in the form of the twiddle factor multiplied by the Montgomery factor, we manage to eliminate half of the multiplications in the process of converting from NTT^{-1} to the integer domain. Finally, for ARMv7-based Cortex A environment, compared to state-of-the-art Kyber PQM4 [1] implementation, our software achieves a performance improvement of 62%, 50%, and 56% in NTT, Point multiplication, and NTT^{-1} , respectively. Also, Compared to state-of-the-art Dilithium PQM4 [2] implementation, our software achieves a performance improvement of 54%, 62%, and 55% in three steps of NTT-based polynomial multiplication, respectively.

B. OUTLINE

The remainder of this article is organized as follows. First, we describe the basic of Crystals-Kyber and Dilithium in Section II and emphasize that the core computation of both schemes is the polynomial multiplication part. We then describe the Number Theoretic Transform (NTT) used by Kyber and Dilithium for polynomial multiplication and analyze the modular arithmetic in NTT-based polynomial multiplication in detail. Section III gives careful consideration to the modular arithmetic analyzed in the previous section for the ARMv7-base Cortex-A series and introduces methods to maximize parallelism. Section IV provide several techniques to accelerate the NTT. We analyze the performance of our proposed techniques in Section VI before concluding the paper in Section V. Additionally, there is an Appendix section VI for a smooth understanding of this paper. The appendix includes the complete scheme algorithms for Kyber and Dilithium, as well as some code proposed in this paper.

II. PRELIMINARY

A. CRYSTALS-KYBER AND CRYSTALS-DILITHIUM

Crystals-Kyber (Kyber) is the only KEM algorithm selected by NIST and its security is based on Module Learning With Error (Module-LWE) problem. Since characteristic of Module-LWE-based Kyber, each element in a matrix and a vector is a polynomial over Ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ where $n = 256$ and $q = 3329$. The public matrix of Kyber exhibits a

TABLE 1. Profiling performance of Kyber and Dilithium reference codes on ODROID-XU4 (ARMv7), NTT means NTT-based polynomial multiplication.

Crystals-Kyber (Kyber768)				
ALG	All	keccak	NTT	others
KeyGen	369,438 (100%)	151,470 (41%)	99,748 (27%)	118,220 (32%)
EnCaps	405,227 (100%)	129,673 (32%)	101,307 (25%)	174,248 (43%)
DeCaps	459,481 (100%)	151,629 (33%)	119,465 (26%)	188,387 (41%)
Crystals-Dilithium (Dilithium3)				
ALG	All	keccak	NTT	others
KeyGen	919,906 (100%)	349,564 (38%)	257,574 (28%)	312,768 (34%)
Sign	3,568,171 (100%)	1,748,404 (49%)	1,106,133 (31%)	713,634 (20%)
Verify	570,998 (100%)	159,879 (28%)	165,589 (39%)	245,529 (43%)

size of $\ell \times \ell$, while the secret vector and noise each possess a size of $\ell \times 1$. For security levels 1, 3, and 5, the corresponding values of ℓ are 2, 3, and 4, respectively. Kyber's Public Key Encryption (PKE) consists of three operations: Key generation, Encryption, and Decryption, and Kyber-KEM utilizes Kyber-PKE with Fujisaki-Okamoto transform for providing IND-CCA2 security. For the complete pseudocode structure of Kyber PKE, please refer to Algorithm 4, 5, and 6 for PKE in Appendix VI-A. Except for the random sampling based on hash function, the core operation of each Kyber-PKE algorithm is either matrix by vector multiplication ($\mathbf{A} \circ \mathbf{s}$) or vector by vector multiplication ($\hat{\mathbf{s}}^T \circ \mathbf{u}$). For example, multiplication with a $\ell \times \ell$ matrix and a $\ell \times 1$ vector requires ℓ^2 polynomial multiplications (vector by vector multiplication requires ℓ polynomial multiplications). In the algorithm description, bold upper-case letters (e.g., $\mathbf{A}$) and bold lower-case letters (e.g., $\mathbf{s}$) denote matrices and vectors, respectively.

In the framework of Kyber, GenMatrixA generates a public matrix $\hat{\mathbf{A}}$, size of $\ell \times \ell$. The result of this process hinges on the input of a 32-bit parameter known as ρ , which leads to the formation of R_q elements via two established methods: SHAKE and Rejection-sampling. Considering the fact that sampling is performed randomly within the R_q , it can be stated that the public matrix $\mathbf{A}$ is derived from the NTT domain. Hence, it is depicted as $\hat{\mathbf{A}}$, eliminating the necessity for explicit NTT. Simultaneously, SampleVec algorithm plays a pivotal role in Kyber, tasked with producing a secret vector and its corresponding noise, both sized $\ell \times 1$. Kyber leverages the Central Binomial Distribution (CBD) in its sampling process, a decision based on the system's predilection for noise and secret vectors that comprise small coefficients. Compress and Decompress serve as pivotal mechanisms in the efficiency of data transmission and optimization of network resources. These procedures sequentially execute

compression and restoration on 8-bit data. The details of the Kyber description can be found in the Kyber specification document [3].

Crystals-Dilithium (Dilithium) is the one of Digital Signature Algorithm selected by NIST. As same as Kyber, its security is based on Module-LWE problem. The polynomial ring used by Dilithium is $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ where $n = 256$ and $q = 2^{23} - 2^{13} + 1$. Similar to Kyber, matrix-vector multiplication is the core operation. Dilithium utilizes Module-LWE problem with Fiat-Shamir With Aborts method for designing DSA. Dilithium has a three security levels, but unlike Kyber (NIST security 1, 3, and 5), it supports NIST security 2, 3, and 5. The size of the public matrix for each security level is (4, 4), (6, 5), and (8, 7), which is not a square matrix unlike Kyber. For the complete pseudocode structure of Dilithium, please refer to Algorithms 7, 8, and 9 in Appendix VI-B. Also, the details of the Dilithium description can be found in the Dilithium specification document [4].

Table 1 presents the performance profiling results, including percentage and clock cycles, for Kyber768 and Dilithium3 on the ARMv7-based Cortex-A series (ODROID-XU4), which is the target platform of this paper. Similar to previous studies [20], [21] on ARMv8-based Cortex-A series, the core function Keccak, associated with Sampling and SHA-3 calls, incurs the highest cost, followed by a significant proportion attributed to NTT-based polynomial multiplication. As the function percentage distribution for short-term (Kyber512 and Dilithium2) and long-term (Kyber1024 and Dilithium5) security closely resembles that of Kyber768 and Dilithium3, these results are omitted in this paper. Generally, since the core part of hash functions is difficult to parallelize, performance enhancement can be expected through parallel invocations of SHA-3 and SHAKE. However, this approach is more suitable for high-performance devices specialized in block/thread parallelization such as GPUs, and may not be desirable for architectures with high thread creation costs. Therefore, in this paper, we primarily focus on NTT-based polynomial multiplication and discuss parallelization strategies.

B. NUMBER THEORETIC TRANSFORM (NTT)

Polynomial multiplication typically has a complexity of $\mathcal{O}(n^2)$ when implemented based on the schoolbook method, but in lattice-based Schemes (LBS) systems, there is a preference for multiplication algorithms with lower complexity. The Number Theoretic Transform (NTT) algorithm is one variation of the Discrete Fourier Transform (DFT) and enables efficient polynomial multiplication in Galois Field. In LBS, NTT can be seen as an evaluation of polynomials and, generally, in a $\mathbb{Z}_q[X]/(X^n - 1)$, n evaluation values are required to determine a polynomial with n coefficients. Hence, the primitive n -th root of unity existing in $\mathbb{Z}_q[X]$ is used to generate these evaluation values. For Cyclic NTT that employs the factor polynomial $X^n - 1$, a primitive n -th root of unity (ω_n) must exist in $\mathbb{Z}_q$ to facilitate NTT usage. Meanwhile, Negacyclic NTT, which uses the factor

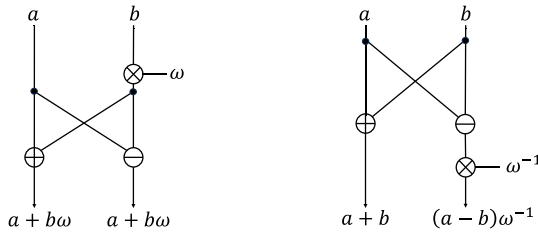


FIGURE 1. (left) CT butterfly, (right) GS butterfly.

polynomial $X^n + 1$, necessitates a primitive $2n$ -th root of unity (ω_{2n}).

Commonly, the Chinese Remainder Theorem (CRT) and the Fast Fourier Transform (FFT) algorithm are applied for an efficient NTT implementation, utilizing only $\mathcal{O}(n \log n)$ operations. Through the CRT algorithm, one can generate an isomorphic mapping of Negacyclic NTT $\mathbb{Z}_q[X]/(X^n - 1) \rightarrow \prod_i \mathbb{Z}_q[X]/(X - \omega_{2n}^{2i+1})$ for $i = 1, 2, \dots, n - 1$, when iterated $\log n$ times, fully decomposes an $n - 1$ degree polynomial into n single coefficients. Each iteration is commonly referred to as an NTT layer, where the result of the j -th layer is the remainder of any polynomial $f \bmod (X^{2^{j-1}} \pm \omega_{2n}^{2i+1})$ for some i . The polynomial elements of Kyber belong to $\mathbb{Z}_q[X]/(X^n + 1)$ with $q = 3329$ and $n = 256$, and since there is no 512-th root of unity within the modulus 3329, Kyber employs an incomplete Negacyclic NTT. In other words, an NTT with 7 layers using the primitive 256-th root of unity $\omega = 17$ is actually implemented, eventually decomposing into a first-degree polynomial. Through the NTT, the original polynomial $X^{256} + 1$ is decomposed as follows:

$$X^{256} + 1 = \prod_{i=0}^{127} (X^2 - \omega_{2n}^{2i+1}) = \prod_{i=0}^{127} (X^2 - \omega_{2n}^{2br_7(i)+1})$$

In here $br_7(i)$ for $i \in \{0, \dots, 127\}$ is the bit reversal of the unsigned 7-bit integer i . Finally, in Kyber, any polynomial $f \in R_q$ is represented as follows by the NTT:

$$f \bmod (X^2 - \omega^{2br_7(0)+1}), \dots, f \bmod (X^2 - \omega^{2br_7(127)+1})$$

Dilithium uses a modulus q that satisfies $512|q - 1$, therefore NTT and NTT^{-1} can be configured with 8 layers. It is composed of the same Negacyclic convolution as Kyber, and because it is partitioned into 8 layers, Point multiplication is relatively simple as a simple coefficient multiplication. To compute the NTT layers efficiently, butterfly operations, typically used in FFT, are utilized. Generally, the Cooley-Tukey (CT) butterfly [32] is used in the NTT and the Gentleman-Sand (GS) butterfly [33] is used in the NTT^{-1} (Please See Figure 1 for illustrations).

Since the split polynomials are in bit-reverse order, rearrangement is required in the implementation. However, by using the CT butterfly, one can transition from the normal order to the bit-reverse order at no additional cost. Similarly, using the GS butterfly in the NTT^{-1} enables us to convert polynomial back to the normal order without any extra cost.

In each butterfly, exactly one multiplication, addition, and subtraction occurs. For implementation efficiency, it is recommended to limit the growth of the coefficients in words around the modulus q (16-bit for Kyber and 32-bit for Dilithium). Therefore, modular multiplication is required, and the different methodologies presented for optimization are compared in the following sections from an efficient implementation perspective on ARMv7-based Cortex-A series.

C. NEON TECHNOLOGY OF ARMV7-BASED CORTEX-A

The ARMv7-based Cortex-M/A series use the 32-bit ARMv7 core with various computational capabilities, where the Cortex-M series is primarily used for microcontrollers and the Cortex-A series for microprocessors. Notably, the ARM Cortex-M4 [15] has been selected as the performance evaluation device for the NIST PQC standard competition. In the case of the Cortex-A series, it includes a powerful parallel engine capable of performing vector operations while providing multiprocessor support. The ARMv7 NEON technology, an advanced Single Instruction Multiple Data (SIMD) extension for ARM Cortex-A processors, is integral for enhancing performance in media-intensive tasks and signal processing. It comprises 32 64-bit registers, which can be paired as 16 128-bit wide units. Supporting diverse integer types and single-precision floating-point, it is adept for a broad range of operations, from basic arithmetic to complex element-wise multiplication. Designed for seamless operation alongside the ARMv7-A main processor, ARMv7 NEON sets a benchmark in versatility, paving the way for subsequent improvements in later architectures like ARMv8-A.

However, when compared to the ARMv8-A NEON, ARMv7 lacks advanced multiplication operations such as polynomial multiplication, complex number multiplication, and rounding versions of multiply-add/subtract instructions. Therefore, it is imperative to utilize the provided vector instructions as efficiently as possible. In this section, we briefly introduce the instructions that will be leveraged in this paper. Comprehensive details regarding all instructions are available in reference [30]. **vmov.sn** moves a n -bit signed scalar to a single lane of the destination register. In the suffixes of the commands, a type is always included. The 's' signifies a signed integer, while 'n' denotes packed bits. Thus, in our implementation of Kyber, we use **s16**, and in Dilithium, we utilize **s32** **vmlal.sn** performs a long multiply accumulate. **vuzp.sn** unzips n -bit elements from the source registers. **vmls.sn** multiplies n -bit elements from the first source register by n -bit elements from the second source register and subtracts the product from the corresponding n -bit elements in the destination register. **vqdmulh.sn** performs a saturated doubling multiply high half, returning the high half of the $2x$ product. **vmul.sn** multiplies n -bit elements from the first source register by n -bit elements from the second source register and writes the product to the destination register. **vhsb.sn** subtracts n -bit

TABLE 2. ARMv7-based NEON Assembly Instructions [30]. r^* means double word unit (result of long multiplication). $[]_h$ ($[]_l$) means result of high (low) multiplication.

Asm	Operand	Operation	cycles
vqdmulh	r, a, b	$r \leftarrow [2 \times (a \times b)]_h$	2
vmul	r, a, b	$r \leftarrow [a \times b]_l$	1
vmull	r^*, a, b	$r^* \leftarrow [a \times b]_l$	2
vhsb	r, a, b	$r \leftarrow (a - b) \gg 1$	2
vadd	r, a, b	$r \leftarrow (a + b)$	2
vmlal	r^*, a, b	$r^* \leftarrow r^* + (a \times b)$	2
vmls	r, a, b	$r \leftarrow r - (a \times b)$	2
vuzp	a, b	de-interleaves of a and b .	2
vtbn	a, b	Transpose each elements	2

elements in the second source register from corresponding elements in the first source register, halves the results. Comprehensive details regarding all instructions are available in reference [30].

Table 2 describes an overview of the ARMv7-based NEON instructions used in this paper, including their cycles. Unfortunately, the public document [30] does not contain timing information per instruction. Implementing ARM assembly as a lower-level C language project allows for relatively simple measurement of the throughput and latency of individual instructions, as execution occurs in order. In the ARMv7-based Cortex-A7 series (ODROID-XU4) board, which is the target environment of this paper, we measure the cycles of the instructions used in our research. While instructions are issued in order, their completion sequence may vary. Therefore, multi-cycle instructions can be executed on different units while independent instructions are issued and completed without interference. We measure the cycles considering the state of the pipeline.

D. PREVIOUS MODULAR ARITHMETIC METHODOLOGY

Three steps in an NTT-based polynomial multiplication require a number of multiplication over $\mathbb{Z}_q$ (Namely, $a \cdot b \bmod q$). In other words, in NTT conversion (using CT butterfly) and NTT^{-1} conversion (using GS butterfly), each coefficient in a polynomial is multiplied with a twiddle factor ω over $\mathbb{Z}_q$ for factoring into smaller polynomials or combining into larger polynomial and in Point multiplication, coefficients of two polynomials of degree 1 are multiplied over $\mathbb{Z}_q$. In Kyber, an element over $\mathbb{Z}_q$ is expressed with a 16-bit data type. Thus, the result of multiplying two elements is 32-bit, and it needs to be reduced by q . In other words, $r = c \bmod q = c - \lfloor c/q \rfloor \cdot q$ ($c = a \cdot b$). The same logic applies to Dilithium. For Dilithium implementation one takes modulus q which is represented as a 32-bit data type, it's important to accumulate the result of the multiplication as 32-bit result. Generally, constant-time implementation is one of the crucial security properties of cryptographic libraries. Initially, security properties were primarily emphasized in terms of the accuracy of cryptographic implementations. However,

recent works have highlighted constant-time implementation as an important issue. For instance, the Keyczar encryption library, which initially used branching (if) for byte-wise comparisons, could be vulnerable to timing attacks for MAC forgery, as pointed out by CodaHale [34]. Furthermore, even cryptographic algorithms designed with constant-time in mind could leak secret values due to branching during implementation, as demonstrated by the Montgomery ladder implementation in OpenSSL 0.9.8o. Specifically, conditional branches used in this implementation could be exploited to steal private keys from TLS servers [35]. Therefore, in research on Modular Arithmetic, it is essential to consider constant-time aspects in both algorithm design and implementation.

As efficient and constant-time reduction algorithms, Montgomery arithmetic [36] and Barrett arithmetic [37] have been widely used. In addition, there is also research that has recently effectively improved Plantard reduction and applied it to Kyber [1]. In the context of LBS using small prime q , the signed Montgomery method (Algorithm 2) and signed Barrett method (Algorithm 1) were proposed [23] and have been applied to the implementations of Kyber and Dilithium. Signed reduction is denoted by $\bmod^\pm q$ and $c \bmod^\pm q$ reduces into $(-\frac{q}{2}, \frac{q}{2})$. The signed representation allows for more flexibility in NTT implementation than the unsigned representation. For example, the extra reduction in the subtraction of CT and GS butterfly disappears.

Algorithm 1 Signed Barrett Reduction [23]

Input: q with $0 < q < \frac{R}{2}$, $2 \nmid q$ and a with $a \in [-\frac{R}{2}, \frac{R}{2})$, where $R = 2^{16}$
Output: $r \equiv a \bmod q$, $r \in [0, q]$

- 1: $v = \left\lfloor \frac{2^{\log(q)-1} \cdot R}{q} \right\rfloor$ ▷ precomputed
- 2: $t = \left\lfloor \frac{av}{2^{\log(q)-1} \cdot R} \right\rfloor$ ▷ signed high product
- 3: $t = tq \bmod R$ ▷ signed low product
- 4: **return** $r = a - t$

Barrett reduction aims to replace the division by q in $\lfloor c/q \rfloor$ with a simpler shift operation. In other words, it can be re-expressed as follows:

$$r = c - \left\lfloor \frac{c}{R} \cdot \frac{R}{q} \right\rfloor \cdot q. \quad (1)$$

In signed representations, the floor function ($\lfloor \cdot \rfloor$) is replaced by the round function ($\lfloor \cdot \rceil$), allowing the range of reduction to extend to negative values. It's important to note that the absolute range remains unchanged. An integer approximation of Barrett reduction for signed representations was proposed in [21] and [23]. Since one operand in the CT and GS Butterfly is a pre-computed twiddle factor (is b below equation), the final form of Barrett reduction is as follows:

$$ab \equiv ab - \left\lfloor \frac{a \left\lfloor \frac{bR}{q} \right\rfloor}{R} \right\rfloor q \pmod{q}. \quad (2)$$

Here, R can be replaced with a shift when implemented as a power of 2. The value of $\lfloor \frac{bR}{q} \rfloor$ can be precomputed as a constant. Therefore, to effectively use Barrett reduction, one of the operands must be a constant value. In other words, in NTT and NTT⁻¹ transformations, we can consider this constant as the Twiddle factor. Algorithm 1 is signed Barrett reduction [23] used by Kyber reference code and note that the input a can be signed value and the result r is in the range of $0 \leq r \leq q$.

Algorithm 2 Signed Montgomery Reduction [23]

It
Input: $c = a \cdot b$ such that $c = c_1 R + c_0$ and $c \in (-\frac{R}{2}q, \frac{R}{2}q)$,
 where $R = 2^{16}$ if $q < 2^{16}$, the odd modulus $q \in (0, \frac{R}{2})$
Output: $r \equiv ab \cdot R^{-1} \pmod{q}$, $r \in (-q, q)$
 1: $m = c_0 \cdot q^{-1} \pmod{\pm R}$ ▷ signed low product
 2: $t_1 = \lfloor m \cdot q/R \rfloor$ ▷ signed high product
 3: $r = c_1 - t_1$
 4: **return** r

The fundamental idea of Montgomery reduction is to find a t such that $c - tq$ is divisible by R . Once t is found, it satisfies $\frac{c-tq}{R} \equiv cR^{-1}$. Similar to Barrett reduction, the general understanding is that the division by q is replaced with a simpler shift operation. In Barrett reduction, one of the multiplication operands must be a pre-computed constant. However, in Montgomery reduction, both operands are free. The initially proposed unsigned Montgomery reduction includes a final reduction process [36]. In the case of signed representations, while the absolute range remains the same, additional subtraction processes are not required since the growth of coefficients does not exceed word size even after NTT computations are completed [23]. The final form is as follows:

$$abR^{-1} \equiv \frac{ab - (ab \cdot q^{-1} \pmod{\pm R}) \cdot q}{R} \pmod{q}. \quad (3)$$

Signed Montgomery reduction is given by Algorithm 2 and the constant R is typically set to l -bit word size such that the modulus q fits in one word. It replaces division with multiplication, shift, and subtraction operations. Note that in the case of subtraction, only the high part ($c_1 - t_1$) of subtraction is required. Since the result r of Algorithm 2 is in the range of $(-q, q)$, it does not need to be unsigned.

Algorithm 3 Improved Plantard Reduction [1]

Input: $c = a \cdot b$, where $a, b \in [-q2^\alpha, q2^\alpha]$, $q < 2^{l-\alpha-1}$,
 and $q' = q^{-1} \pmod{\pm 2^{2l}}$
Output: $r = c(-2^{-2l}) \pmod{\pm q}$, where $r \in (-\frac{q}{2}, \frac{q}{2})$
 1: $r = \lfloor ([cq']_{2l})^l + 2^\alpha q \rfloor$
 2: **return** r

The basic idea of Plantard reduction [38] is similar to Montgomery reduction, aiming to find a value of t such that $(tq - ab)$ is divisible by $R = 2^{2l}$. Ultimately, it should satisfy $(tq - ab)^{-2l} \equiv ab(-2^{-2l}) \pmod{q}$. Recently, an improved

version of Plantard multiplication was proposed in [1] for Cortex-M4, extending the method to a signed system and expanding the input range (cf. Algorithm 3). The improved Plantard reduction method provides a range 2^α times larger ($[-q^2 2^{2\alpha}, q^2 2^{2\alpha}]$) compared to Montgomery reduction, while maintaining half of the output range ($(-\frac{q}{2}, \frac{q}{2})$). Leveraging Cortex-M4's **smulwb** and **smlabb** instructions, and when one of the operands is constant, only two multiplications occur during the modular multiplication process.

III. PROPOSED MODULAR ARITHMETIC FOR ARMV7-BASED CORTEx-A SERIES

In this section, we propose an implementation methodology for selecting modular arithmetic and considering the NEON instruction set to implement Kyber and Dilithium on the ARMv7-based Cortex-A series.

A. CONSIDERATION FOR ARMV7 NEON TECHNOLOGY

First, we examine the arithmetic on the Cortex-M4, which uses the same core as the most popular and target architecture of this paper, the ARMv7-based Cortex-A series. The state-of-the-art implementation proposed in [1] is an improvement of the original Plantard multiplication, restructured for Signed representation (cf. Algorithm 3). Unfortunately, the improved Plantard reduction can be implemented using the **smulwb** and **smlabb** instructions of the Cortex-M4. Specifically, for Kyber, it requires support for both 16×32 -bit multiplications, and precisely the upper 32-bit of the result need be returned. However, ARMv7-based NEON Technology does not support multiplication for different word sizes, and even if expanded to 32-bit multiplication, there is no instruction to return the upper result, making it impossible to use the improved Plantard reduction. Moreover, especially in the case of Dilithium, since 64-bit multiplication is required, the number of coefficients that need to be packed is halved. Thus, the improved Plantard reduction is not effective in the ARMv7-based Cortex-A series.

Next, we turn our attention to the NEON environment based on ARMv8 core. In the ARMv8 environment, [21] proposed an advanced approximation, building on the foundational Barrett reduction. Through his research, Becker illustrated a linkage between Barrett and Montgomery multiplications, conclusively identifying the result as a 32-bit value. This breakthrough facilitated vectorization when deploying NEON in the ARMv8 context. However, the efficiency of this method largely hinges on the robust capabilities of the **smull1** and **smull2** instruction, for Montgomery multiplication, which provide high and low returns of extended multiplication. Additionally, due to the differences in the number of registers and the instruction set between the ARMv8 environment and ARMv7 environment, it is very challenging to fully port the Barrett multiplication proposed in [21] to the ARMv7-based Cortex-A series. As mentioned in [21], their target SIMD ISA, which includes M-Profile Vector Extension (MVE), the Scalable Vector Extension 2 (SVE2) (16-bit lanes only), and AVX2, is different from the

```
int16_t montgomery_reduce(int32_t a) {
    int16_t t
    t = (int16_t) a * qinv
    t = (a - (int32_t)t * q) >> 16
    return t}

```

LISTING 1. Ref Montgomery C code of Kyber [3].

```
int16_t our_montgomery(int32_t a) {
    int16_t t
    t = (int16_t) a * qinv
    t = {(2*a-(int32_t)t*q*2) / 2} >> 16
    return t}

```

LISTING 2. Our Montgomery C code for Kyber.

```
.macro mc_mont_mul
    vqdmulh.s16 r, a, b
    vmul.s16 t, a, bqinv
    vqdmulh.s16 t, t, q
    vhsb.s16 r, r, t
.endm

```

LISTING 3. Our Montgomery ASM code for Kyber.

ARMv7-based Cortex-A series. In other word, regrettably, the NEON instruction set available in ARMv7 don't offer the same latitude as their ARMv8 counterparts. Considering these limitations, in this paper, we aim to introduce a method tailored for Montgomery multiplication and Barrett reduction that aligns with the specifications of the NEON extension in ARMv7 environment.

B. MONTGOMERY MULTIPLICATION

Listing 2 shows a modification of Montgomery reduction tailored for the ARMv7-based Cortex-A series. Listing 1 shows a reference montgomery reduction of Kyber [3]. Unlike the ARMv8-based neon engine, the 32-bit neon engine lacks upper word multiplication or lower word multiplication capabilities (**smull** and **smull2**). Instead, it only supports **vqdmulh** (vector saturating doubling multiply returning high-half). Therefore, we adapt the Montgomery reduction formula to leverage this specific instruction. Failing to utilize the **vqdmulh** instruction would necessitate an additional instruction for long word multiplication. The assembly code, with montgomery multiplication of Kyber, incorporating the **vqdmulh** instruction and **vhsb** (vector halving subtract) is shown in Listing 3. In other words, the Montgomery multiplication can be rewritten follows:

$$abR^{-1} \equiv \frac{2ab - (2ab \cdot q^{-1} \bmod^{\pm} R) \cdot q}{2R} \pmod{q}. \quad (4)$$

The general operation of Montgomery multiplication based on the Kyber standard involves invoking Montgomery reduction (cf. Listing 1) on the 32-bit product result after a 16-bit long multiplication. However, long multiplication disrupts the consistency of parallelization logic in vector

implementations. This issue, a significant concern, has already been addressed in AVX2 environment studies [13], [23]. Consistent with previous research, to maintain parallelization, it's essential to use high/low multiplication instead of long multiplication, preserving packed words. Consequently, employing the approach in Algorithm 2 allows for replacing the division by $2R$ in the equation above with an implicit shift operation. Additionally, the subtraction in the numerator can be calculated using only the higher product. In essence, as with Listing 3, Montgomery multiplication can be implemented with just four instructions.

In contrast to the reference implementation where multiplication is followed by reduction, we implement it in a combined manner (cf. Listing 3), that is, as Montgomery multiplication. Where b is a twiddle factor, we assume that $bqinv$ can be precomputed. This approach boasts the advantage of allowing concurrent multiplication and reduction for 8/4 coefficients for Kyber/Dilithium. Given that accumulation result by multiply-long instruction is not required, it conservatively uses registers. The capability to handle concurrent multiplication for 8/4 (Kyber/Dilithium) coefficients stems from the transformation of Montgomery multiplication for using **vqdmulh** and **vhsb** instruction (refer to Listing 2 and 3). The code currently presented in this section for listing 1, 2, and 3 are all implementations for Kyber. For Dilithium's implementation, the approach is the same. Only the word size needs to be modified to convert to Dilithium code. The code for Dilithium can be found in the Appendix VI-C.

C. BARRETT REDUCTION

Listing 5 presents a variation of Barrett reduction specifically designed for the ARMv7-based Cortex-A series. Listing 4 shows a reference barrett reduction of Kyber [3]. The goal, for Kyber, is to enhance performance while retaining 8 16-bit coefficients within a 128-bit vector register. To achieve this, we adjust the parameter of Barrett constant to $R = 2^{16}$, enabling the utilization of an implicit shift. The finalized assembly code is provided in Listing 6. The assembly code for Barrett reduction employs six NEON instructions for Kyber. Our Barrett reductions are executed concurrently for eight coefficients. Although adjusting $R = 2^{26}$ to $R = 2^{16}$ slightly alters the range of the reduction results, it does not materially impact the overall functioning of Kyber.

For instance, in the case of Kyber using Montgomery multiplication (cf. Algorithm 2), the output range is $(-q, q)$, and even after accumulating up to 7 layers, the polynomial coefficients grow to be up to 16-bit following the NTT implementation. Therefore, Barrett reduction must be performed after NTT to achieve correct Point multiplication. In Barrett reduction, the chosen approximation ($R = 2^t$) generally increases in value as t increases, resulting in a more precise range for reduction. This means that the final modular result can be closer to the desired value. A larger t provides more precise intermediate results, enhancing the accuracy of the final result, but also increases

```

int16_t barrett_reduce(int16_t a) {
    int16_t t
    const int16_t v = ((1<< 26) + q/2)/q
    t = (int32_t)v*a + (1<< 25) >> 26
    t *= q
    t = a - t
    return t}

```

LISTING 4. Ref Barrett C code of Kyber [3].

```

int16_t our_barrett(int16_t a) {
    int16_t t
    const int16_t v = ((1<< 16) + q/2)/q
    t = (int32_t)v*a + (1<< 15) >> 16
    t *= q
    t = a - t
    return t}

```

LISTING 5. Our Barrett C code for Kyber.

```

.macro mc_barrett_reduce
    vmov.s32 t, #32768
    vmov.s32 v, #32768
    vmlal.s16 t, a0, barrett_const
    vmlal.s16 v, a1, barrett_const
    vuzp.s16 t, v
    vmls.s16 r, v, q
.endm

```

LISTING 6. Our Barrett ASM code.

computational complexity. Conversely, a decrease in t leads to reduced accuracy, meaning the final modular result can be further from the original value. In our implementation using $R = 2^{16}$ for implicit shift operations in the NEON engine, we map to a larger range $(-\frac{3}{4}q, \frac{3}{4}q)$ compared to the typical Kyber's Barrett reduction ($R = 2^{26}$), which maps to $(-\frac{q}{2}, \frac{q}{2})$. After NTT and NTT^{-1} , it is ensured that each coefficient lies within $(-\frac{3}{4}q, \frac{3}{4}q)$, allowing all of them to be mapped to $[0, q)$ during the final packing process, where they are converted into positive numbers. In other words, for our implementation, choosing $R = 2^{16}$ slightly increases the reduction range, but this does not hinder the process of polynomial multiplication based on NTT.

In the case of reference implementation of Dilithium, Dilithium uses a variation (reduce32) of Barrett reduction. Therefore, unlike Kyber, where addition occurs before multiplication, our goal is to efficiently implement the reference code using ARMv7-based NEON instructions, rather than changing the coefficients for an implicit shift. In the ARMv7 NEON environment, reduce32 can be implemented with just four instructions. The related code can be found in Listing 10, 11 in the Appendix VI-C.

TABLE 3. Register scheduling on NEON vector register.

Reg	Used	Kyber (8×16 -bit)	Dilithium (4×32 -bit)
R ₀	zetas	$(\zeta_0 \parallel \dots \parallel \zeta_0)$	$(\zeta_0 \parallel \zeta_0 \parallel \zeta_0 \parallel \zeta_0)$
...	...	...	...
R ₃	zetas	$(\zeta_3 \parallel \dots \parallel \zeta_3)$	$(\zeta_3 \parallel \zeta_3 \parallel \zeta_3 \parallel \zeta_3)$
R ₄	coeffs	$(c_0 \parallel \dots \parallel c_7)$	$(c_0 \parallel c_1 \parallel c_2 \parallel c_3)$
...	...	...	...
R ₇	coeffs	$(c_{96} \parallel \dots \parallel c_{103})$	$(c_0 \parallel c_1 \parallel c_2 \parallel c_3)$
...	...	...	...
R ₈	coeffs	$(c_{128} \parallel \dots \parallel c_{135})$	$(c_0 \parallel c_1 \parallel c_2 \parallel c_3)$
...	...	...	...
R ₁₁	coeffs	$(c_{224} \parallel \dots \parallel c_{231})$	$(c_0 \parallel c_1 \parallel c_2 \parallel c_3)$
R ₁₂	q, q^{-1}	$(q \parallel q \parallel q^{-1} \parallel q^{-1})$	$(q \parallel q \parallel q^{-1} \parallel q^{-1})$
R ₁₃	tmp ₀	$(\dots \parallel \dots \parallel \dots)$	$(\dots \parallel \dots \parallel \dots)$
...	...	...	...
R ₁₅	tmp ₂	$(\dots \parallel \dots \parallel \dots)$	$(\dots \parallel \dots \parallel \dots)$

IV. NTT-BASED POLYNOMIAL IMPLEMENTATION

In this section, we describe our methodology for implementing NTT-based polynomial multiplication utilizing the modular arithmetic proposed in previous Section III (Montgomery multiplication and Barrett reduction tailored for ARMv7-based Cortex-A series).

A. PACKING ON NEON REGISTER

ARMv7 NEON features 16 vector registers, each with a size of 128 bits. Data packing can be done in 16-bit, 32-bit, or 64-bit units. In a similar approach to the previous research on ARMv8-based Cortex-A series [21], we pack the coefficients of Kyber, which has 12-bit modulus $q = 3329$, in 16-bit units. This means that a total of 8 coefficients of Kyber's polynomials are computed simultaneously in one register. Out of the 16 registers, 8 are used to store the polynomial coefficients, and 4 are used to store the twiddle factors (here, twiddle factors denoted by ζ). The remaining 4 registers are utilized for storing q, q^{-1} , and intermediate values necessary for Montgomery multiplication and Barrett reduction. Since the modulus $q = 8380417$ of Dilithium is 23-bit, we proceed with coefficient packing in 32-bit units. For implementing Dilithium, while we use the registers in the same way as for Kyber, the number of coefficients and twiddle factors packed is half compared to Kyber. Table 3 shows the register scheduling for our work. The order of coefficients is arranged considering the merging of layers.

B. FORWARD NTT

We implement NTT using CT butterflies. To minimize memory access costs, we adopt a merging strategy, extensively researched in [2] and [13]. This strategy harnesses the powerful load and store instructions inherent to the ARM architecture. Typically, the implementation of each layer in Forward NTT follows the approach depicted in Figure 2 (right). To compute the first layer for polynomial $\mathbf{c} = c_0 + c_1X + \dots + c_{255}X^{255} \in \mathbb{Z}_q[X]/(X^n + 1)$, coefficients starting

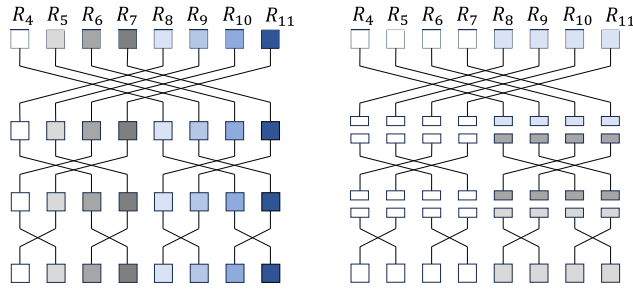


FIGURE 2. (left) Merging strategy, (right) General method.

from c_0 are sequentially stored in registers R_4 to R_7 (colored white), while coefficients starting from c_{128} are sequentially stored in registers R_8 to R_{11} (colored light blue). Once the computation of the first layer is complete, the results are stored in memory, necessitating a reload of coefficients for the computation of the second layer. This sequential placement of coefficients in the registers requires memory access for each layer. Consequently, to apply a merging strategy, as illustrated in Figure 2 (left), coefficients for the first layer are loaded into registers R_4 to R_{11} following the arrangement specified in Table 3. This method enables the computation of up to three layers in one go, without the need for additional memory access after the initial load of coefficients in first layer.

1) KYBER

In Cortex-M4 environment, One can load/store four coefficients into the register simultaneously, enabling us to perform butterfly operations across multiple layers at once. The number of registers available determines the maximum layers we can merge. Considering the need to retain constants for modular arithmetic in the register, we have found merging up to four layers on the Cortex-M4 to be efficient, as detailed in (3-3-1 or 4-3) by [2]. In the ARMv7-based NEON environment, we merge three layers, followed by two, then another two, out of the seven layers in NTT (3-2-2). The reason one can merge 4-3 layers on Cortex-M4 is because it uses Floating-Point Unit (FPU) registers. If ARMv7-based Cortex-A series used FPU registers, it would conflict with the gcc vectorization option at compile time, and there are not enough vector registers, so we use a more generic merge strategy.

2) DILITHIUM

We approach the implementation of Dilithium using the same methodology as Kyber. In Dilithium, which uses a 23-bit modulus $q = 8380417$, each polynomial coefficient can only store four in a single vector register. Since the NTT in Dilithium proceeds up to 8 layers, we use a 3-3-2 strategy to merge each layer.

C. POINT MULTIPLICATION

1) KYBER

In Kyber, there is no primitive 512-th root of unity, necessitating that the NTT and NTT^{-1} of Kyber are conducted only

for 7 layers. In matrix-vector multiplication, for coefficients of the polynomial elements of the public matrix ($\mathbf{a} = a_0 + a_1x$) and the secret vector ($\mathbf{s} = s_0 + s_1x$), point-wise multiplication from NTT elements perform $(a_0 + a_1x) \cdot (s_0 + s_1x)$. Therefore, during the point multiplication, it is essential to perform multiplication for first-degree polynomials, and considering the multiplication with the twiddle factor, a total of five coefficient multiplications occur. Previous studies [21], [23] have demonstrated that in high-performance environments where the cycle times for multiplication and addition instructions are equal, the schoolbook method proves to be faster than the Karatsuba method. Consistent with previous research conducted on ARMv8-based NEON [21], we implement Kyber's point multiplication using the schoolbook approach.

2) DILITHIUM

Conversely, since Dilithium performs $\text{NTT}/\text{NTT}^{-1}$ up to 8 layers, we have implemented single coefficient multiplication utilizing all registers. Therefore, four point multiplications are performed simultaneously in one register.

D. INVERSE NTT

1) KYBER

We implement NTT^{-1} with GS butterflies. Similarly to NTT, we merge the seven layers in reverse order, combining them as two, then another two, and finally (2-2-3). During the last phase of NTT^{-1} , we integrate the conversion process to the integer domain into the final layer, and pre-multiply the Montgomery multiplication of half the coefficients by the twiddle factor. This strategy eliminates the need for 128 Montgomery multiplications. In the reference code of Kyber, Barrett reduction is called at each layer of NTT^{-1} . We calculate a precise output range for each layer and, call Barrett reduction only at the 3rd and 5th layers when using the GS butterfly in NTT^{-1} . In other words, during each merging stage, Barrett reduction is called 640 times less (5×128 times) than in the reference implementation.

2) DILITHIUM

In the case of Dilithium, which uses a 23-bit modulus $q = 8380417$, the coefficients are packed in 32-bit units. Therefore, the accumulated values in each layer do not exceed 32 bits even after the completion of all 8 layers. Therefore, unlike Kyber, additional reductions are not required during the progress of each layer. Our Dilithium implementation merges each layer in a 2-2-3 pattern. The multiplication of the Montgomery factor for conversion to the integer domain is invoked in the computation of the final layer. Similar to Kyber, no additional memory costs are incurred.

V. EXPERIMENTAL RESULTS

A. BENCHMARKING SETUP

We have selected the ODROID-XU4 device, which features an ARM Cortex-A7 MPU, as our target board.

TABLE 4. Cycle counts (cc) for NTT-based polynomial multiplication on ARMv7-based device.

Crystals-Kyber						
Software	Device	Language	gcc option	NTT (cc)	PointMul (cc)	NTT ⁻¹ (cc)
This work	ODROID-XU4 (32-bit Cortex-A7)	Asm	-O3 (implies vectorize)	1,718 (+62%)	1,200 (+50%)	2,038 (+56%)
PQM4 [1] (-)	STM32F407G (32-bit Cortex-M4)	Asm	-O3 (using FPU register)	4,474	2,422	4,684
Ref [3]	ODROID-XU4 (32-bit Cortex-A7)	C	-O3 (implies vectorize)	11,692	8,713	13,278
NEON NTT [21]	Raspberry Pi 4B+ (64-bit Cortex-A72)	Asm	-O3 (implies vectorize)	1,200	952	1,338
Crystals-Dilithium						
Software	Device	Language	gcc option	NTT (cc)	PointMul (cc)	NTT ⁻¹ (cc)
This work	ODROID-XU4 (32-bit Cortex-A7)	Asm	-O3 (implies vectorize)	3,735 (+54%)	738 (+62%)	3,824 (+55%)
PQM4 [2] (-)	STM32F407G (32-bit Cortex-M4)	Asm	-O3 (using FPU register)	8,093	1,955	8,415
Ref [4]	ODROID-XU4 (32-bit Cortex-A7)	C	-O3 (implies vectorize)	12,532	8,072	14,328
NEON NTT [21]	Raspberry Pi 4B+ (64-bit Cortex-A72)	Asm	-O3 (implies vectorize)	2,241	1,378	2,821

For compilation, we employ **gcc 11.3.0** and the following compilation options: **-O3 -mcpu=native -mfpu=neon -ftree-vectorize**. For comparative evaluation, we utilize the most recent PQM4 code and the reference code using vectorize option for compiling. The most recent PQM4 code incorporate the Better-accumulation and Asymmetric-multiplication techniques from PQM4. In a PQM4 scenario, we use a non-vectorized implementation of the keccak algorithm, but highly optimized assembly code (tailored for Cortex-M4 [39]), ensuring a benchmark comparison without altering the codebase. Both the reference and the proposal presented in this paper undergo comparison using the keccak implementation, optimized with neon-vectorized assembly [39]. In other words, the comparison group for Kyber consists of the implementation proposed in [1], integrated into PQM4 (marked as PQM4 [1]), and the reference implementation (marked as Ref [3]). The comparison group for Dilithium consists of the implementation proposed in [2], integrated into PQM4 (marked as PQM4 [2]), and the reference implementation (marked as Ref [4]). Each PQM4 implementation uses the FPU register, hence the vectorize option is not used during the build, while the reference implementation does use the vectorize option.

B. PERFORMANCE OF NTT

Table 4 shows the implementation results for NTT-based polynomial multiplication. Although the Ref [3] and [4] implementation can use the vectorization option at compile time, it is written in C language and shows lower

performance compared to the highly optimized implementation of PQM4 [1], [2]. However, our implementation (marked This work) shows high performance improvement compared to the state-of-the-art PQM4 implementation results in each step of NTT-based polynomial multiplication. This shows that the modular arithmetic proposed in this paper for ARMv7-based Cortex-A series is clearly effective in performing NTT-based polynomial multiplication process with maximum parallelism. For ARMv7-based Cortex A environment, Compared to PQM4 [1], our Kyber implementation achieves a performance improvement of 62%, 50%, and 56% in NTT, Point multiplication, and NTT⁻¹, respectively. Also, Compared to PQM4 [2], our Dilithium implementation achieves a performance improvement of 54%, 62%, and 55% in NTT, Point multiplication, and NTT⁻¹, respectively. Taking a closer look, in the case of Kyber, computations are performed on 8 coefficients at once in vector registers. Ideally, this should result in an eightfold performance improvement compared to the Reference [3]. However, vector instructions inherently require more cycles compared to pure ARM instructions, leading to additional delays at each layer. In this paper, we mitigate these delays using Merging techniques. As a result, for Kyber, only an additional cost of about 300 cycles is incurred. The PQM4 implementation reduces Modular Multiplication to 2 cycles using Plantard multiplication. While our multiplication takes 8 cycles (4 instructions), leveraging Merging allows us to handle 8 coefficients simultaneously, justifying approximately a 62% performance improvement. In the case of Dilithium, computations are performed on 4 coefficients

TABLE 5. Cycle counts (cc) for Kyber and Dilithium on ARMv7-based device.

Crystals-Kyber							
Software	Device	Language	gcc option	API	Kyber512 (cc)	Kyber768 (cc)	Kyber1024 (cc)
This work	ODROID-XU4 (32-bit Cortex-A7)	Asm	-O3 (implies vectorize)	KeyGen	110k (+51%)	185k (+50%)	294k (+47%)
				EnCpas	139k (+48%)	230k (+43%)	348k (+41%)
				DeCaps	133k (+57%)	221k (+52%)	339k (+48%)
PQM4 [1]	STM32F407G (32-bit Cortex-M4)	Asm	-O3 (using FPU register)	KeyGen	430k	702k	1,119k
				EnCaps	526k	861k	1,314k
				DeCaps	472k	780k	1,211k
Ref [3] (-)	ODROID-XU4 (32-bit Cortex-A7)	C	-O3 (implies vectorize)	KeyGen	223k	369k	552k
				EnCaps	265k	405k	590k
				DeCaps	309k	459k	655k
NEON NTT [21]	Raspberry Pi 4B+ (64-bit Cortex-A72)	Asm	-O3 (implies vectorize)	KeyGen	62k	99k	156k
				EnCaps	81k	1275k	192k
				DeCaps	76k	120k	184k
Crystals-Dilithium							
Software	Device	Language	gcc option	API	Dilithium2 (cc)	Dilithium3 (cc)	Dilithium5 (cc)
This work	ODROID-XU4 (32-bit Cortex-A7)	Asm	-O3 (implies vectorize)	KeyGen	424k (+17%)	790k (+14%)	1,256k (+16%)
				Signing	1,279 (+41%)	2,087k (+41%)	2,652k (+41%)
				Verifying	436k (+24%)	738k (+21%)	1,260k (+20%)
PQM4 [2]	STM32F407G (32-bit Cortex-M4)	Asm	-O3 (using FPU register)	KeyGen	1,596k	2,827k	4,829k
				Signing	4,093k	6,623k	8,803k
				Verifying	1,572k	2,692k	4,707k
Ref [4] (-)	ODROID-XU4 (32-bit Cortex-A7)	C	-O3 (implies vectorize)	KeyGen	512k	920k	1,498k
				Signing	2,158k	3,568k	4,506k
				Verifying	571k	928k	1,574k
NEON NTT [21]	Raspberry Pi 4B+ (64-bit Cortex-A72)	Asm	-O3 (implies vectorize)	KeyGen	270k	516k	782k
				Signing	649k	1,089k	1,437k
				Verifying	272k	447k	765k

simultaneously. Performance analysis can be conducted similarly to Kyber. Due to differences in processors, there still exists a performance gap between the latest research on the ARMv8-based Cortex-A series [1], [2], [3], [4], and [21], implementations; however, we have managed to narrow this gap.

C. PERFORMANCE OF KYBER

Table 5 shows the overall performance of Kyber and Dilithium. In the comparison of NTT, the highly optimized implementation of PQM4 demonstrated higher performance compared to the Reference implementation. However, from the perspective of the entire scheme, it is observed that the reference implementation with the vectorize option enabled shows higher performance than the PQM4 implementation in both Kyber and Dilithium. Therefore, we choose the

reference implementation compiled with the vectorization option as our comparison target. The reason for this includes the fact that both the Reference and our software use vectorized code for the keccak, which accounts for a significant portion of Kyber's computational workload. As mentioned earlier, this is a choice made for a fair performance comparison. Our Kyber512 implementation achieves a performance improvement of 51%, 48%, and 57% in KeyGen, EnCapsulation, and DeCapsulation, respectively. For Kyber768, Our work achieves a performance improvement of 50%, 43%, and 52% in KeyGen, EnCapsulation, and DeCapsulation, respectively. Finally, Our Kyber1024 implementation achieves a performance improvement of 47%, 41%, and 48% in KeyGen, EnCapsulation, and DeCapsulation, respectively. When compared with the ARMv8-based Cortex-A series, despite optimization efforts conducted

on the ARMv7-based Cortex-A, there remains a clear performance difference. This is primarily due to differences in the supported instruction sets and a twofold variance in the number of vector registers. Nevertheless, we have significantly narrowed the performance gap compared to the previous results.

D. PERFORMANCE OF DILITHIUM

Similar to the comparison in previous section, which compare implementation of Kyber, we analysis a performance comparison of Dilithium against the vectorized Reference code rather than the PQM4 [2] code (Please see Table 5). Our Dilithium2 implementation achieves a performance improvement of 17%, 41%, and 24% in KeyGen, Signing, and Verifying, respectively. For Dilithium3, Our work achieves a performance improvement of 14%, 41%, and 21% in KeyGen, Signing, and Verifying, respectively. Finally, Our Dilithium5 implementation achieves a performance improvement of 16%, 41%, and 20% in KeyGen, Signing, and Verifying, respectively. In the case of Dilithium, the reason for the higher performance improvement of the Signing process compared to KeyGen and Verifying is that Dilithium has a process to regenerate the signature when an abort occurs. In other words, there is a performance difference between each process compared to Kyber. The primary reason for the performance enhancement lies in the vectorized NTT-based multiplication.

VI. CONCLUSION

In this paper, we presented a parallel processing solution for NTT-based polynomial multiplication in Kyber and Dilithium, selected for standardization by NIST, utilizing the NEON extension set in an ARMv7 environment. To our knowledge, this work the first implementation on ARMv7-based Cortex-A series for achieving quantum security. Notably, we introduced modified parameter sets and operational methods to maintain the consistency of parallel logic in modular multiplication and single-coefficient reduction. As a result, parallel processing became more efficient due to implicit shifts facilitating faster Barrett reduction, and Montgomery multiplication allowed for parallelism throughout the multiplication and reduction without additional registers and instructions. Additionally, by adopting advanced optimization strategies, we performed merging in each NTT and NTT^{-1} , and eliminated 128 multiplications in the integer domain conversion process in the final layer of NTT^{-1} . Ultimately, compared to the latest optimized PQM4 implementations and parallelized Reference implementations, our approach achieved significant performance improvements in each scheme. To summarize, Our implementation of Kyber512 (Kyber768 and Kyber1024) shows a performance improvement of 51% (50% and 47%), 48% (43% and 41%), and 57 (52% and 48%)% in Key Generation, Encapsulation, and Decapsulation process, respectively, compared to the vectorized compiling option set in the reference implementation [3]. Similarly, our

implementation of Dilithium2 (Dilithium3 and Dilithium5) achieves performance improvements of 17% (14% and 16%)%, 41% (41% and 41%)%, and 24% (21% and 20%)% in Key Generation, Signing, and Verifying process, respectively, when compared to the vectorized compiling option set in the reference. Our methodology underscores the effectiveness of adopting parallelized implementation strategies utilizing the NEON engine in Cortex-A series, which use the same ARMv7 core as the Cortex-M4, over the PQM4 implementations. This work not only contributes to the field of quantum-safe cryptography on ARM architectures but also opens avenues for further exploration in optimizing post-quantum cryptography algorithms for other architectures, considering the balance between performance and security. We believe that our proposed approach to modular arithmetic is not limited to Kyber and Dilithium, but can be applied to algorithms that require modular arithmetic, such as Saber, NTRU, and Falcon. Based on this, we plan to apply our methodology to algorithms in the Falcon and NIST PQC-DSA competitions in the ARMv7 environment.

APPENDIX

A. KYBER.CPAPKE ALGORITHM

Algorithm 4 Kyber.CPAPKE KeyGen [1], [3]

Output: $pk = (\hat{\mathbf{b}}, \rho)$, $sk = \hat{\mathbf{s}}$

- 1: $seed \leftarrow \{0, \dots, 255\}^{32}$
- 2: $\rho, \sigma \leftarrow \text{SHAKE256}(64, seed)$
- 3: $\hat{\mathbf{A}} \leftarrow \text{GenMatrixA}(\rho)$
- 4: $\mathbf{s} \leftarrow \text{SampleVec}(\sigma, 0)$
- 5: $\mathbf{e} \leftarrow \text{SampleVec}(\sigma, 1)$
- 6: $\hat{\mathbf{b}} \leftarrow \hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s}) + \text{NTT}(\mathbf{e})$
- 7: **return** $pk = (\hat{\mathbf{b}}, \rho)$, $sk = \hat{\mathbf{s}}$

Algorithm 5 Kyber.CPAPKE Enc [1], [3]

Input: $pk = (\hat{\mathbf{b}}, \rho)$, message μ in R_q , seed $coin \in \{0, \dots, 255\}^{32}$

Output: Ciphertext $(\mathbf{u}', h)$

- 1: $\hat{\mathbf{A}} \leftarrow \text{GenMatrixA}(\rho)$
- 2: $\mathbf{s}' \leftarrow \text{SampleVec}(coin, 0)$
- 3: $\mathbf{e}' \leftarrow \text{SampleVec}(coin, 1)$
- 4: $\mathbf{e}'' \leftarrow \text{SampleVec}(coin, 2)$
- 5: $\hat{\mathbf{t}} \leftarrow \text{NTT}(\mathbf{s}')$
- 6: $\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{t}}) + \mathbf{e}'$
- 7: $\hat{\mathbf{v}} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{v}}^T \circ \hat{\mathbf{t}}) + \mathbf{e}'' + \mu$
- 8: **return** $(\mathbf{u}' = \text{Compress}(\mathbf{u}), h = \text{Compress}(\mathbf{v}'))$

Algorithm 6 Kyber.CPAPKE Dec [1], [3]

Input: Ciphertext $c = (\mathbf{u}', h)$, secret key $sk = \hat{\mathbf{s}}$

Output: Message $\mu \in R_q$

- 1: $\mathbf{u} \leftarrow \text{Decompress}(\mathbf{u}')$
- 2: $\mathbf{v}' \leftarrow \text{Decompress}(h)$
- 3: **return** $\mu = \mathbf{v}' - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u}))$

```
int32_t montgomery_reduce(int64_t a) {
    int32_t t
    t = (int32_t) a * qinv
    t = (a - (int64_t)t * q) >> 32
    return t}

```

LISTING 7. Ref Montgomery C code of Dilithium [4].

```
int32_t our_montgomery(int64_t a) {
    int32_t t
    t = (int32_t) a * qinv
    t = {(2*a-(int64_t)t*q*2) / 2} >> 32
    return t}

```

LISTING 8. Our Montgomery C code for Dilithium.

B. DILITHIUM DIGITAL SIGNATURE ALGORITHM

Algorithm 7 Dilithium KeyGen [4], [20]

Output: $pk = (\rho, \mathbf{t}_1)$, $sk = (\rho, K, tr, s_1, s_2, \mathbf{t}_0)$

- 1: $\rho, K \leftarrow \{0, 1\}^{256}$
- 2: $(s_1, s_2) \in S_\eta^\ell \times S_\eta^k := \text{ExpandA}(K)$
- 3: $\mathbf{A} \in R_q^{k \times \ell} := \text{ExpandA}(\rho)$
- 4: $\mathbf{t} := \mathbf{A}s_1 + s_2$
- 5: $(\mathbf{t}_1, \mathbf{t}_0) := \text{Power2Round}(\mathbf{t}, d)$
- 6: $tr \in \{0, 1\}^{384} := \text{CRH}(\rho \parallel \mathbf{t}_1)$
- 7: **return** $pk = (\rho, \mathbf{t}_1)$, $sk = (\rho, K, tr, s_1, s_2, \mathbf{t}_0)$

Algorithm 8 Dilithium Signature Generation [4], [20]

Input: Secret key sk and a message M

Output: Signature $\sigma = (\mathbf{z}, \mathbf{h}, c)$

- 1: $\mathbf{A} \in R_q^{k \times \ell} := \text{ExpandA}(\rho)$
- 2: $\mu \in \{0, 1\}^{384} := \text{CRH}(tr \parallel M)$
- 3: $\kappa := 0, (\mathbf{z}, \mathbf{h}) := \perp$
- 4: **while** $(\mathbf{z}, \mathbf{h}) := \perp$ **do**
- 5: $y \in S_{\gamma_1-1}^\ell := \text{ExpandMask}(K \parallel \mu \parallel \kappa)$
- 6: $\mathbf{w} := \mathbf{A}y$
- 7: $\mathbf{w}_1 := \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$
- 8: $c \in B_{60} := \text{H}(\mu \parallel \mathbf{w}_1)$
- 9: $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
- 10: $(\mathbf{r}_0, \mathbf{r}_1) := \text{Decompose}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$
- 11: **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$ **then** $(\mathbf{z}, \mathbf{h}) := \perp$
- 12: **else**
- 13: $\mathbf{h} := \text{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 - c\mathbf{t}_0, 2\gamma_2)$
- 14: **if** $\|\mathbf{c}\mathbf{t}_0\| \geq \gamma_2$ or the # of 1's in $\mathbf{h}$ is greater than w **then**
- 15: $(\mathbf{z}, \mathbf{h}) := \perp$
- 15: $\kappa = \kappa + 1$
- 16: **return** $\sigma = (\mathbf{z}, \mathbf{h}, c)$

Algorithm 9 Dilithium Verification [4], [20]

Input: Public key pk , message M and signature σ .

Output: Signature Verification

- 1: $\mathbf{A} \in R_q^{k \times \ell} := \text{ExpandA}(\rho)$
- 2: $\mu \in \{0, 1\}^{384} := \text{CRH}(\text{CRH}(\rho \parallel \mathbf{t}_1) \parallel M)$
- 3: $\mathbf{w}'_1 := \text{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$
- 4: **return** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $c := \text{H}(\mu \parallel \mathbf{w}'_1)$ and # of 1's in $\mathbf{h}$ is $\leq w$

C. DILITHIUM CODE FOR ARMV7-BASED CORTEX-A

ACKNOWLEDGMENT

This work was partly supported by the Institute of Information and communications Technology Planning and Evaluation (IITP) Grant by the Korean Government through Ministry of Science and ICT (MSIT) (No.RS-2022-00207416, A study on PQC optimization and security protocol migration

```
.macro mc_mont_mul
    vqdmulh.s32 r, a, b
    vmul.s32 t, a, bqinv
    vqdmulh.s32 t, t, q
    vhsb.s32 r, r, t
.endm

```

LISTING 9. Our Montgomery ASM code for Dilithium.

```
int32_t reduce32(int32_t a) {
    int32_t t
    t = (a + (1 << 22)) >> 23
    t = a - t*q
    return t}

```

LISTING 10. Ref reduce32 C code of Dilithium [4].

```
.macro mc_reduce32
    vmov.s32 t, #0x400000
    vadd.s32 t, src, t
    vshr.s32 t, t, #23
    vmls.s32 src, t, q
.endm

```

LISTING 11. Our reduce32 ASM code for Dilithium.

to neutralize advanced quantum attacks in Beyond 5G-based next-generation IoT computing environments, 50%) and was partly supported by IITP Grant by MSIT (No. RS-2023-00225201, Development of Control Rights Protection Technology to Prevent Reverse Use of Military Unmanned Vehicles, 50%).

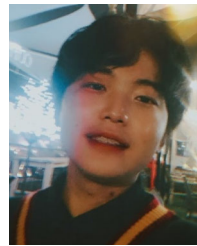
CONFLICT OF INTEREST

All authors have no conflict of interest

REFERENCES

- [1] J. Huang, J. Zhang, H. Zhao, Z. Liu, R. C. C. Cheung, Ç. K. Koç, and D. Chen, "Improved plantard arithmetic for lattice-based cryptography," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 1, pp. 614–636, Aug. 2022.
- [2] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels, "Faster Kyber and Dilithium on the Cortex-M4," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, Jun. 2022, pp. 853–871.
- [3] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, (2020). *CRYSTALS–Kyber*. [Online]. Available: <https://pq-crystals.org/kyber/>
- [4] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, (2020). *CRYSTALS–Dilithium*. [Online]. Available: <https://pq-crystals.org/dilithium/>
- [5] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, (2020). *Falcon Submission to the NIST Post-Quantum Cryptography Standardization*. [Online]. Available: <https://falcon-sign.info/>

- [6] A. Hulsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kolbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens. (2020). *SPHINCS⁺*. [Online]. Available: <https://sphincs.org/>
- [7] (2023). *OpenSSL Manual Page*. [Online]. Available: <https://www.openssl.org/docs/man3.0/man7/crypto.html>
- [8] (2023). *OpenSSH Manual Page*. [Online]. Available: <https://www.openssh.com/manual.html>
- [9] (2023). *Mbed-TLS*. [Online]. Available: <https://github.com/Mbed-TLS/mbedtls>
- [10] (2013). *AVR-crypto-Lib*. [Online]. Available: <https://github.com/cantora/avr-crypto-lib>
- [11] C. A. Lara-Nino, A. Diaz-Perez, and M. Morales-Sandoval, "Elliptic curve lightweight cryptography: A survey," *IEEE Access*, vol. 6, pp. 72514–72550, 2018.
- [12] E. Alkim, Y. Alper Bilgin, M. Cenk, and F. Gérard, "Cortex-M4 optimizations for R, MLWE schemes," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 1, pp. 336–357, Jun. 2020.
- [13] C.-M.-M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "NTT multiplication for NTT-unfriendly rings: New speed records for saber and NTRU on cortex-M4 and AVX2," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 1, pp. 159–188, Feb. 2021.
- [14] A. Abdulrahman, J.-P. Chen, Y.-J. Chen, V. Hwang, M. J. Kannwischer, and B.-Y. Yang, "Multi-moduli NTTs for saber on cortex-M3 and cortex-M4," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 1, pp. 127–151, Nov. 2021.
- [15] ARM. (2010). *Cortex-M4 Tech. Reference Manual*. [Online]. Available: <https://developer.arm.com/documentation/ddi0439/b/>
- [16] M. J. Kannwischer, R. Petri, J. Rijneveld, P. Schwabe, and K. Stoffelen. *PQM4: Post-quantum Crypto Library for the ARM Cortex-M4*. Accessed: Mar. 3, 2024. [Online]. Available: <https://github.com/mupq/pqm4>
- [17] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "Improving software quality in cryptography standardization projects," in *Proc. IEEE Eur. Symp. Secur. Privacy Workshops*, Jun. 2022, pp. 19–30.
- [18] V. Hwang, Y. Kim, and S. C. Seo. (2023). *Barrett Multiplication for Dilithium on Embedded Devices*. [Online]. Available: <https://eprint.iacr.org/2023/1955>
- [19] D. O. C. Greconici, M. J. Kannwischer, and D. Sprenkels, "Compact dilithium implementations on cortex-M3 and cortex-M4," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 1, pp. 1–24, Dec. 2020.
- [20] Y. Kim, J. Song, T.-Y. Youn, and S. C. Seo, "Crystals-dilithium on ARMv8," *Secur. Commun. Netw.*, vol. 2022, pp. 1–12, Feb. 2022.
- [21] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: Faster dilithium, kyber, and saber on cortex-A72 and apple M1," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 1, pp. 221–244, Nov. 2021.
- [22] Y. Kim, J. Song, and S. C. Seo, "Accelerating falcon on ARMv8," *IEEE Access*, vol. 10, pp. 44446–44460, 2022.
- [23] G. Seiler. (2018). *Faster AVX2 Optimized NTT Multiplication for Ring-LWE Lattice Cryptography*. [Online]. Available: <https://eprint.iacr.org/2018/039>
- [24] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: FrodoKEM, NewHope, and kyber," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 575–586, Mar. 2021.
- [25] S. C. Seo, "SIKE on GPU: Accelerating supersingular isogeny-based key encapsulation mechanism on graphic processing units," *IEEE Access*, vol. 9, pp. 116731–116744, 2021.
- [26] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of kyber on cortex-M4," in *Progress in Cryptology—AFRICACRYPT*. Cham, Switzerland: Springer, 2019, pp. 209–228.
- [27] J. W. Bos, J. Renes, and A. Sprenkels, "Dilithium for memory constrained devices," in *Progress in Cryptology—AFRICACRYPT*. Cham, Switzerland: Springer, 2022, pp. 217–235.
- [28] D. T. Nguyen and K. Gaj, "Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists," in *Proc. Int. Conf. Post-Quantum Cryptogr.*, Jul. 2021, pp. 234–254.
- [29] S. Kaiser, Md. S. Haq, A. S. Tosun, and T. Korkmaz, "Container technologies for ARM architecture: A comprehensive survey of the state-of-the-art," *IEEE Access*, vol. 10, pp. 84853–84881, 2022.
- [30] (2021). *ARMv7-A Archit. Reference Manual*. [Online]. Available: <https://developer.arm.com/documentation/ddi0406/cb>
- [31] (2021). *ARM Archit. Reference Manual, ARMv8, for ARMv8-A Archit. Profile*. [Online]. Available: <https://developer.arm.com/documentation/ddi0487/gb/?lang=en>
- [32] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, p. 297, Apr. 1965.
- [33] W. M. Gentleman and G. Sande, "Fast Fourier transforms: For fun and profit," in *Proc. Fall Joint Comput. Conf.*, 1966, pp. 563–578.
- [34] C. Hale. (2009). *A Lesson in Timing Attacks (or, Don't Use MessageDigest.Isequals)*. [Online]. Available: <https://codahale.com/a-lesson-in-timing-attacks/>
- [35] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Proc. European Symposium Research Computer Security*, 2011, p. 12.
- [36] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, p. 519, Apr. 1985.
- [37] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Conf. Theory Appl. Cryptograph. Techn.* Cham, Switzerland: Springer, 1986, pp. 311–323.
- [38] T. Plantard, "Efficient word size modular arithmetic," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 3, pp. 1506–1518, Jul. 2021.
- [39] P. L. Montgomery. *Extended Keccak Code Package*. [Online]. Available: <https://github.com/XKCP/XKCP>
- [40] *Post-Quantum Cryptography Standardization Project*. Accessed: Mar. 3, 2024. [Online]. Available: <https://csrc.nist.gov/Projects/post-quantum-cryptography>



YOUNGBEOM KIM (Student Member, IEEE) received the B.S. degree from the Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul, South Korea, in 2021, and the M.S. degree in financial information security from Kookmin University, in 2023, where he is currently pursuing the Ph.D. degree. His research interests include the optimization of cryptographic algorithms, post-quantum cryptography implementation, module validation programs, and SW/HW co-design for IoT devices.



SEUNGYONG YOON (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Chungnam National University, Daejeon, South Korea, in 1999 and 2001, respectively. Since 2001, he has been a Principal Researcher with the Electronics and Telecommunications Research Institute (ETRI), Daejeon. He has also been developing technologies for network security gateway, anti-DDoS systems, mobile trusted modules, embedded OS, lightweight security protocol, and IoT device security. His current research interests include cryptographic protocol, post-quantum cryptography, key hiding and management, device authentication, and the IoT security technologies.



SEOG CHUNG SEO (Member, IEEE) received the B.S. degree in information and computer engineering from Ajou University, Suwon, South Korea, in 2005, the M.S. degree in information and communications from Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, in 2007, and the Ph.D. degree from Korea University, Seoul, South Korea, in 2011. He was a Research Staff Member with the Samsung Advanced Institute of Technology (SAIT) and Samsung DMC Research and Development Center, from September 2011 to April 2014. He was a Senior Research Member of the Affiliated Institute, ETRI, South Korea, from 2014 to 2018. He is currently an Assistant Professor with Kookmin University, South Korea. His research interests include public-key cryptography, its efficient implementations on various IT devices, cryptographic module validation programs, network security, and data authentication algorithms.

...