




Article

HAETAE on ARMv8

Minjoo Sim ¹, Minwoo Lee ² and Hwajeong Seo ^{2,*}

¹ Department of Information Computer Engineering, Hansung University, Seoul 02876, Republic of Korea; minjoos9797@hansung.ac.kr

² Department of Convergence Security, Hansung University, Seoul 02876, Republic of Korea; 1771397@hansung.ac.kr

* Correspondence: hwajeong@hansung.ac.kr; Tel.: +82-760-8033

Abstract: In this work, we present the highly optimized implementation of the HAETAE algorithm, submitted to the second round of the Korean Post-Quantum Cryptography (KpqC) competition and to the first round of NIST's additional post-quantum standardization for digital signatures on 64-bit ARMv8 embedded processors. To the best of our knowledge, this is the first optimized implementation of the HAETAE algorithm on 64-bit ARMv8 embedded processors. We apply various optimization techniques to enhance the multiplication operations in the HAETAE algorithm. We utilize parallel operation techniques involving vector registers and NEON (Advanced SIMD technology used in ARM processors) instructions of ARMv8 embedded processors. In particular, we achieved the best performance of the HAETAE algorithm on ARMv8 embedded processors by applying all the state-of-the-art NTT (Number Theoretic Transform) implementation techniques. Performance improvements of up to $3.07\times$, $3.63\times$, and $9.15\times$ were confirmed for NTT, Inverse-NTT, and pointwise Montgomery operations (Montgomery multiplication used in modular arithmetic), respectively, by applying the state-of-the-art implementation techniques, including the proposed techniques. As a result, we achieved a maximum performance improvement of up to $1.16\times$ for the key generation algorithm, up to $1.14\times$ for the signature algorithm, and up to $1.25\times$ for the verification algorithm.

Keywords: HAETAE; 64-bit ARMv8 processors; post-quantum cryptography; software implementation; parallel implementation; KpqC competition



Citation: Sim, M.; Lee, M.; Seo, H. HAETAE on ARMv8. *Electronics* **2024**, *13*, 3863. <https://doi.org/10.3390/electronics13193863>

Academic Editors: Lixin Wang, Jianhua Yang, Radhouane Chouchane and Linqiang Ge

Received: 12 August 2024

Revised: 13 September 2024

Accepted: 24 September 2024

Published: 29 September 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The advent of quantum computers is propelling significant technological advancements but simultaneously threatens the security of modern cryptographic systems. Quantum computers, as theorized by Richard Feynman, leverage the principles of quantum mechanics to operate [1]. These machines can execute quantum algorithms such as Shor's and Grover's algorithms. Shor's algorithm [2] enables efficient factorization of large numbers, while Grover's algorithm [3] accelerates search processes. These algorithms have the potential to compromise the mathematical structures that underpin contemporary cryptographic systems. For instance, public key encryption schemes relying on prime factorization could become vulnerable in the quantum era. The pace of quantum computing advancements is rapid; for example, IBM unveiled the Osprey quantum computer with 433 qubits in 2022, followed by the Condor with 1121 qubits in 2023 [4]. Some studies suggest that breaking RSA-2048, one of the most widely used public key encryption algorithms, might require around 372 qubits under ideal conditions [5].

At the same time, the proliferation of Internet of Things (IoT) devices has introduced new cybersecurity challenges [6]. As IoT devices become more prevalent in industrial and consumer applications, they are increasingly responsible for processing sensitive information and managing critical systems. Given their widespread integration into daily life and essential infrastructure, securing these devices against evolving threats is becoming

paramount. The rapid growth of IoT devices highlights the need for robust cryptographic solutions that can protect against future attacks.

Quantum computing poses a significant risk to the cryptographic algorithms that protect IoT devices, including smart home systems, wearable health monitors, and industrial sensors [7–11]. Many of these devices rely on legacy cryptographic techniques implemented on processors like the ARM Cortex-M series, RISC-V processors, and ARM Cortex-A series, which may be rendered insecure by quantum advances. The challenge is exacerbated by the resource limitations of IoT devices, which often have constrained processing power and memory. Ensuring secure and efficient encryption in the quantum era requires the development and deployment of post-quantum cryptographic algorithms that are both resistant to quantum attacks and compatible with the operational constraints of IoT environments.

In response to these emerging challenges, the National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography (PQC) Standardization Competition [12]. Similarly, since February 2022, the Korean Post-Quantum Cryptography (KpqC) Competition has been taking place to develop a distinct post-quantum cryptography algorithm [13]. Research efforts focusing on optimizing NIST PQC standardization and its candidate algorithms on various embedded processors, such as 8-bit AVR, 32-bit Cortex-M4, and 64-bit ARMv8, are ongoing [14–22]. Similar optimization efforts should be pursued for the candidate algorithms in the KpqC Competition.

HAETAE [23] is a digital signature algorithm that advanced to the second round of the KpqC competition and was also selected for the first round of NIST's additional post-quantum standardization. This paper presents a high-performance implementation of HAETAE on 64-bit ARMv8 processors, utilizing advanced optimization techniques to achieve exceptional performance.

The remainder of this paper is structured as follows. Section 2 describes the KpqC Competition, the HAETAE algorithm, the HAETAE algorithm on Cortex-M4, the target 64-bit ARMv8 processor, and related works. Section 3 describes the proposed method. Section 4 shows a performance comparison. Finally, Section 5 describes the conclusion of this paper and future work.

1.1. Contribution

1.1.1. Applying State-of-the-Art NTT Implementation Techniques

We introduce a data reordering and pre-computation technique specifically for the HAETAE algorithm on ARMv8 processors. By integrating these techniques with cutting-edge NTT implementations, we achieve optimal HAETAE performance on ARMv8 processors. Our implementation utilizes 128-bit vector registers and NEON instructions of ARMv8 processors, resulting in performance improvements of up to $3.07\times$ for NTT, $3.63\times$ for inverse NTT, and $9.15\times$ for pointwise Montgomery operations. Consequently, our approach enhances the performance of key generation by up to $1.16\times$, signature generation by up to $1.14\times$, and verification by up to $1.25\times$.

1.1.2. First Implementation of the HAETAE on 64-bit ARMv8 Processors Using NEON Instructions

To the best of our knowledge, we present the first implementation of the HAETAE utilizing 64-bit ARM processors' vector registers. ARMv8 processors, known for their high performance, are widely used in various devices, including notebooks, mobile phones, and tablets. Our optimized implementation aims to be applicable across many fields by targeting widely-used processors. We believe this work will serve as a valuable resource for researchers assessing HAETAE performance and will support further advancements in the field.

2. Background

2.1. KpqC Contest

In 2022, the KpqC Competition was initiated in Korea to identify and standardize a quantum-resistant algorithm. The competition aims to stimulate the development of post-quantum secure cryptographic algorithms within the Korean cryptographic community, addressing the challenges presented by the progress in quantum computing. The KpqC competition established four primary evaluation criteria. The first is cryptographic security. Each candidate algorithm must meet rigorous security standards, and the accompanying whitepaper must provide evidence of the defined security properties. The second criterion is efficiency. With security being a non-negotiable requirement, the algorithms are further assessed based on their efficiency once security is ensured. The third criterion is usability. Algorithms should be adaptable across various platforms and systems; otherwise, their usability would be limited, causing inconvenience. Thus, broad platform and system support are crucial to enhance the algorithm's versatility and practicality. The final criterion is originality, to select the most innovative algorithm as the Korean standard for cryptography.

In the first round of the competition, 16 candidate algorithms were chosen, comprising 7 PKE/KEM algorithms and 9 digital signature algorithms. The results of Round 1 were announced in December 2023, and Round 2 is currently in progress for the selected algorithms. A total of eight algorithms advanced to Round 2, including four PKE/KEM algorithms and four digital signature algorithms.

Research related to the KpqC competition, including side-channel analysis, security evaluation, security analysis, and implementation analysis, is actively being conducted [24–29].

In particular, the HAETAE algorithm was selected as one of the KpqC Round 2 candidate algorithms.

2.2. HAETAE

The HAETAE (Hyperball bimodal module rejection signature scheme) is a lattice-based signature algorithm designed to solve the challenges of the Learning With Errors (LWE) and Short Integer Solution (SIS) problems [23]. In July 2023, it was selected as one of the additional candidate algorithms in NIST's PQC competition [30]. The algorithm employs the "Fiat-Shamir with Aborts" paradigm, using rejection sampling to convert signatures that rely on sensitive information into publicly simulatable signatures.

The Fiat-Shamir with Aborts paradigm was introduced in lattice-based cryptography by Lyubashevsky [31,32]. In this paradigm, the verification key consists of a pair of matrices $(A, T = A \cdot S \bmod q)$, where A is a uniformly random matrix modulo of some integer q , and S is a small-magnitude matrix that constitutes the secret key. A signature for a message M is constructed as an integer vector \mathbf{z} , where $\mathbf{z} = \mathbf{y} + S \cdot \mathbf{c}$, with \mathbf{y} being a randomly selected small-magnitude vector and $\mathbf{c} = H(A \cdot \mathbf{y} \bmod q, M)$ being a small-magnitude challenge. Rejection sampling is applied to ensure that the resulting signature distribution is independent of the secret key. The verification algorithm checks whether the vector \mathbf{z} is within an acceptable bound and whether $\mathbf{c} = H(A \cdot \mathbf{z} - T \cdot \mathbf{c} \bmod q, M)$.

The HAETAE was designed with inspiration from the "Fiat-Shamir with Aborts" signature scheme CRYSTALS-DILITHIUM, which was selected by NIST for post-quantum cryptography standardization. However, HAETAE exhibits two significant differences. Instead of using a unimodal distribution as in CRYSTALS-DILITHIUM, it employs a bimodal distribution for rejection sampling, similar to the BLISS signature scheme. Additionally, the HAETAE utilizes hyperball uniform distributions instead of discrete hypercube uniform distributions. It relies solely on continuous Gaussian samplers and allows for extensive parallelization, even in the part where it calls these samplers.

To improve signature compactness, as discussed in [33], the choice of sampling and rejection distributions plays a crucial role in determining the signature size. While the Dilithium scheme utilizes discrete uniform distributions over hypercubes, which simplifies implementation, these distributions are suboptimal in terms of achieving smaller signature

sizes. In contrast, the HAETAE adopts a different approach, sacrificing some ease of implementation to achieve more compact signatures.

However, it should be noted that the HAETAE's implementation is more complex than CRYSTALS-DILITHIUM because it involves real-number calculations and is implemented with floating-point numbers when sampling from the hyperball. Despite this complexity, it offers several advantages.

The HAETAE provides signatures that are 30% to 40% smaller than those of CRYSTALS-DILITHIUM at the same security level. Verification sizes are 20% smaller than those of CRYSTALS-DILITHIUM. Although it involves complex computations, optimized versions of the HAETAE are predicted to run at the same speed as CRYSTALS-DILITHIUM. The HAETAE offers smaller signature sizes compared to CRYSTALS-DILITHIUM, along with improved implementation aspects and better protection against side-channel attacks when compared to FALCON and Mitaka. Moreover, most operations within the HAETAE are relatively straightforward and amenable to constant-time implementation and masking. The HAETAE parameters are shown in Table 1.

Table 1. Parameters of HAETAE (n is ring dimension, q is fully split modulo integer, η is infinity norm of the secret key, and τ is hamming weight of the binary challenge).

Scheme	n	q	η	τ	Security Level	Verify Key (bytes)	Secret Key (bytes)	Signature (bytes)
HAETAE120	256	64,513	1	58	2	992	1376	1474
HAETAE180	256	64,513	1	80	3	1472	2080	2349
HAETAE260	256	64,513	1	128	5	2080	2720	2948

2.3. HAETAE on Cortex-M4

The HAETAE was implemented on a Cortex-M4 environment using the STM32F4-Discovery board. The primary components affecting the computational efficiency of the HAETAE are Keccak, NTT, and Hyperball sampling. Keccak and NTT can be replaced with existing optimized code. Specifically, for NTT, minor modifications to the constants allow for the reuse of Dilithium's implementation. Keccak was replaced with a portable Keccak implementation, which provided a simple yet significant performance improvement. Additionally, performance was further enhanced by optimizing polynomial arithmetic and the Gaussian sampler.

2.3.1. Polynomial Arithmetic Optimization on Cortex-M4

The HAETAE algorithm uses a modulus of $q = 64,513 = 0xFC01$, which is an unsigned 16-bit prime and possesses a 512th root of unity. Elements fully reduced in \mathbb{Z}_q can be stored efficiently in either the upper or lower 16 bits of a 32-bit register. However, this storage technique is not directly applicable for arithmetic operations. When performing lazy reduction or addition, 17 bits are needed to store the result, and a total of 18 bits are required when combining additions with lazy multiplication. Unfortunately, HAETAE's modulus is incompatible with Plantard multiplication, preventing the use of this method.

Most post-quantum cryptographic algorithms use primes with 13 bits or fewer. In such scenarios, two signed 16-bit values can be packed into a 32-bit register for more efficient coefficient storage. This approach is beneficial when coefficients are written once and used without modification, as 16-bit storage can be more efficient. However, there is no instruction that takes 16-bit unsigned integers and produces a 17-bit output suitable for modified Montgomery reduction. Dilithium's Montgomery reduction, which uses $R = 2^{32}$, requires three instructions for implementation. Consequently, HAETAE struggles to outperform this method and operates with 32-bit coefficients, which is consistent with its use in other polynomial arithmetic operations beyond the expansion of the public key polynomial.

Given the similarities between HAETAE and Dilithium, NTT optimizations for M4, developed by Abdulrahman et al. [34], were reused in HAETAE. This optimization reduced the NTT cycle count from 37,506 to 8047 cycles, resulting in a $4.6\times$ performance improvement compared to the reference implementation. Similarly, the inverse NTT cycle count was reduced from 43,116 to 8369 cycles, providing a $5.1\times$ speed improvement over the reference implementation.

2.3.2. Gaussian Sampler Optimization on Cortex-M4

The Gaussian sampler in HAETAE is primarily composed of two elements: the CDT sampler, which is responsible for sampling the most significant bits, and a fixed-point exponential function that is used in the rejection step. Both of these components were optimized to enhance performance. The CDT sampler operates by comparing 16-bit uniformly sampled random values against precomputed threshold tables, generating 1s and 0s based on the comparisons. This was parallelized using SIMD instructions on the Cortex-M4 processor, specifically utilizing `uadd16`, `usub16`, and `sel`. The `uadd16` instruction adds corresponding 16-bit values into a 32-bit register, while `usub16` performs subtraction. The `sel` instruction selects values from two registers based on a condition, facilitating efficient branching within loops. These instructions optimize both memory access and loop performance, leading to a $3.9\times$ speed improvement compared to the reference implementation.

The exponential function was approximated through polynomial evaluation using Horner's method. The reference implementation performed fixed-point arithmetic with 48 fractional bits, involving 64-bit integer values and 64×64 to 128-bit multiplication. Since the Cortex-M4 processor does not natively support 128-bit multiplication, each multiplication was broken down by splitting the value 'a' into higher bits ($ah = a \gg 24$) and lower bits ($al = a - (ah \ll 24)$). The results were then accumulated using the `smlal` instruction, which handles a 32×32 to 64-bit multiply-accumulate operation. This optimization resulted in a $2.9\times$ speedup.

Table 2. Arrangement specifier combinations of vector register; Q: is the second and upper half specifier. Ta, Tb is an arrangement specifier.

<Q>	-	2	-	2	-	2
Ta	8h	8h	4s	4s	2d	2d
Tb	8b	16b	4h	8h	2s	4s

2.4. Target Processor: 64-bit ARMv8 Architecture

ARM is a high-performance embedded processor based on the Instruction Set Architecture (ISA). The ARMv8-A architecture supports both 32-bit AArch32 and 64-bit AArch64 modes, ensuring backward compatibility with older systems. ARMv8-A provides 32 vector registers, each 128-bits wide, labeled from `v0` to `v31`, alongside 31 general-purpose registers, each 64-bits wide, labeled from `x0` to `x30`.

The general-purpose registers can also function as 32-bit registers, denoted as `w0` to `w30`. Vector registers are capable of parallel operations, allowing values stored within them to be processed in different unit sizes. Specifically, these units include byte (8-bit), half-word (16-bit), single-word (32-bit), and double-word (64-bit) divisions, as illustrated in Figure 1. Table 2 details the arrangement specifier combinations used when applying vector instructions. Vector instructions, often referred to as NEON, enable parallel computations using the vector registers.

The list of instructions utilized in the proposed implementations is presented in Table 3 [35].

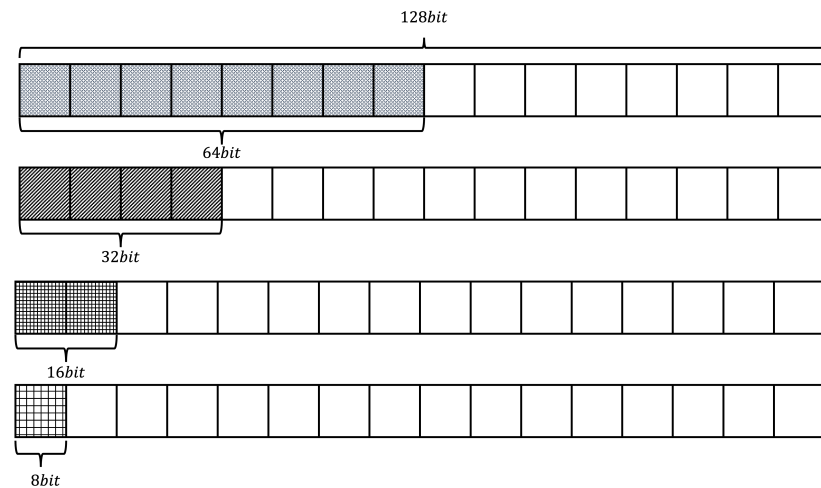


Figure 1. Register packing of vector registers.

Table 3. Summarized instruction set of ARMv8 for HAETAE; X_d , V_d : destination register (general, vector), X_n , V_n , V_m : source register (general, vector, vector), V_t : transferred vector register.

asm	Operands	Description	Operation
ADD	$V_d.T, V_n.T, V_m.T$	Add	$V_d \leftarrow V_n + V_m$
LD1	$V_t.T, [X_n]$	Load multiple single-element structures	$V_t \leftarrow [X_n]$
LD1R	$V_t.T, [X_n]$	Load single 1-element structure and replicate to all lanes (of one register).	$V_t.T \leftarrow [X_n]$
MOV	$X_d, \#imm$	Move (immediate)	$X_d \leftarrow \#imm$
MOV	$V_d.T, V_n.T$	Move (vector)	$V_d \leftarrow V_n$
MOVI	$V_t.T, \#imm$	Move immediate (vector)	$V_t \leftarrow \#imm$
MUL	V_d, V_n, V_m	Multiply	$V_d \leftarrow V_n \times V_m$
SMULL	$V_d.Ta, V_n.Tb, V_m.Tb$	Signed Multiply Long (lower half)	$V_d \leftarrow V_n \times V_m$
SMULL2	$V_d.Ta, V_n.Tb, V_m.Tb$	Signed Multiply Long (upper half)	$V_d \leftarrow V_n \times V_m$
SMLSL	$V_d.Ta, V_n.Tb, V_m.Tb$	Signed Multiply-Subtract Long (lower half)	$V_d \leftarrow V_n \times V_m$
SMLSL2	$V_d.Ta, V_n.Tb, V_m.Tb$	Signed Multiply-Subtract Long (upper half)	$V_d \leftarrow V_n \times V_m$
RET	$\{X_n\}$	Return from subroutine	Return
SHL	$V_d.T, V_n.T, \#shift$	Shift Left immediate (vector)	$V_d \leftarrow V_n \ll \#shift$
SSHR	$V_d.T, V_n.T, \#shift$	Signed Shift Right and immediate (vector)	$V_d \leftarrow V_n \gg \#shift$
ST1	$V_t.T, [X_n]$	Store multiple single-element structures from one, two, three, or four registers	$[X_n] \leftarrow V_t$
SUB	$X_d, X_n, \#imm$	Subtract immediate	$X_d \leftarrow X_n - \#imm$
SUB	V_d, V_n, V_m	Subtract	$V_d \leftarrow V_n - V_m$
REV32	$V_d.T, V_n.T$	Reverse elements in 32-bit words	$V_d \leftarrow V_n$ of Reverse
REV16	$V_d.T, V_n.T$	Reverse elements in 16-bit words	$V_d \leftarrow V_n$ of Reverse
CBNZ	$Wt, Label$	Compare and Branch on Nonzero	Go to Label
ZIP1	$V_d.T, V_n.T, V_m.T$	Zip vectors primary	$V_d \leftarrow V_n[even], V_m[even]$ $V_d \leftarrow V_n[odd], V_m[odd]$
ZIP2	$V_d.T, V_n.T, V_m.T$	Zip vectors secondary	$V_d \leftarrow V_n[even], V_m[even]$ $V_d \leftarrow V_n[odd], V_m[odd]$
XTN, XTN2	$V_d.Tb, V_n.Ta$	Extracted Narrow	$V_d \leftarrow V_n$
SQDMULH	$V_d.T, V_n.T, V_m.T$	Signed saturating Doubling Multiply returning High half	$V_d \leftarrow 2 \times V_n \times V_m$
SHSUB	$V_d.T, V_n.T, V_m.T$	Signed Halving Subtract	$V_d \leftarrow (V_n - V_m)/2$
TRN1	$V_d.T, V_n.T, V_m.T$	Transpose vectors primary	$V_d \leftarrow V_n[even], V_m[even]$ $V_d \leftarrow V_n[odd], V_m[odd]$

2.5. Previous Implementations of Post Quantum Cryptography on 64-bit ARMv8 Processors

Kim et al. performed an optimization of CRYSTALS-DILITHIUM on 64-bit ARM Cortex-A processors, a digital signature algorithm that has been chosen as part of the NIST PQC standardization competition [19]. The optimizations were focused on ARMv8-A architecture, aiming to reduce memory accesses, enable parallel processing, and efficiently use ARM/NEON instructions. Their approach involved optimizing Number Theoretic Transform (NTT), inverse NTT, and pointwise multiplication. For instance, memory access was minimized by using techniques such as merging and register-holding, allowing operations to be performed within registers. This significantly reduced the load and store instructions, improving performance in embedded environments. Moreover, by utilizing ARM-specific instructions like SMSUBL, they were able to optimize the butterfly method, which processed two coefficients in parallel, thus speeding up signed multiplications and Montgomery reduction. Additionally, NEON instructions, including SMULL, XTN, and SMLSL, were used to handle multiple coefficients at once, improving performance by operating on four coefficients simultaneously in a 128-bit vector register. As a result, their implementation achieved substantial performance improvements, with gains of $43.83\times$, $113.25\times$, and $41.92\times$ for KeyGen, Sign, and Verify operations compared to the reference CRYSTALS-DILITHIUM implementation for security level 3.

Becker et al. implemented optimizations of NTT and Barrett multiplication on ARMv8-A using NEON instructions [21]. They combined Montgomery multiplication with Barrett reduction, making it particularly effective for modular multiplication. Their optimizations also included interleaved multi-stage butterfly techniques, which resulted in significant speed-ups for Saber and Kyber cryptographic operations. For example, on the Apple M1 processor, their NTT implementation showed a $2.1\times$ and $1.9\times$ improvement for matrix–vector multiplication in Kyber and Saber, respectively, compared to the reference implementations.

Kwon et al. optimized the FrodoKEM algorithm, a post-quantum public-key encryption and key encapsulation mechanism, for 64-bit ARMv8 processors [36]. Their work featured parallel matrix multiplication and the use of an AES accelerator for AES-128 encryption. They introduced a method capable of generating 80 elements of the output matrix simultaneously, applying this to FrodoKEM640. By utilizing a 128-bit vector register and NEON instructions, their implementation outperformed a previous C-based implementation by up to $10.22\times$.

Additionally, Kwon et al. optimized the Rainbow signature scheme on ARMv8 processors [37]. Rainbow is a multivariate-based signature algorithm and was a finalist in the NIST PQC competition. They developed a tower-field multiplication method utilizing a lookup table to enhance performance. Their implementation was also resistant to timing attacks and achieved significant performance improvements on various processors, including Apple A13 Bionic, Cortex-A72, and Apple M1. Their method demonstrated a $428.73\times$ speedup for finite field multiplications and a $114.16\times$ improvement for the Rainbow signature scheme over previous reference implementations.

Sim et al. implemented Classic McEliece, a code-based key encapsulation algorithm, on ARMv8 [38]. They introduced parallelization techniques, leveraging the commutative property of certain operations. Their optimizations resulted in a maximum performance gain of $2.82\times$ compared to previous C implementations.

3. Proposed Method

CRYSTALS-DILITHIUM is a digital signature algorithm selected for NIST PQC standardization, and many researchers have conducted research to optimize CRYSTALS-DILITHIUM [39]. Among them, there are studies that optimally implemented CRYSTALS-DILITHIUM on the ARMv8 processors [19,21].

Building on this body of work, we have adapted similar techniques used in the optimized NTT implementation of the HAETAE on the Cortex-M4 for our ARMv8 implementation. By leveraging the modifications made to the previously optimized Dilithium's

NTT implementation, we were able to achieve significant performance improvements on the ARMv8 platform.

3.1. Optimized Implementation of NTT Utilizing NEON Instructions

Montgomery reduction is a technique used in modular arithmetic to efficiently compute the modulo operation of large integers [40]. By replacing the division operation in modular multiplication with a sequence of simpler operations, Montgomery reduction improves computational efficiency. This method is particularly beneficial in polynomial arithmetic, where element-wise modular polynomial multiplication reduction helps optimize polynomial multiplication.

The Number Theoretic Transform (NTT) is a mathematical transformation akin to the Fast Fourier Transform (FFT) but specifically designed to operate within finite fields or rings [41,42]. This specialization makes NTT particularly suitable for modular arithmetic operations, which are fundamental in cryptographic applications. Unlike the FFT, which operates over real or complex numbers, NTT works with integers modulo, a prime or composite number, thereby enhancing its applicability in contexts where modular arithmetic is essential.

Mathematically, for a polynomial $P(x)$ of degree $n - 1$ over a finite field \mathbb{F}_q or a ring \mathbb{Z}_N , the NTT is defined as:

$$P(\omega^k) = \sum_{i=0}^{n-1} P(x_i) \cdot \omega^{ik} \quad (1)$$

where ω denotes a primitive n -th root of unity modulo q or N , and ω^k represents the k -th power of this root. This formulation allows for the efficient transformation of polynomial coefficients into a domain where polynomial multiplication can be performed more quickly.

The NTT can also be represented in matrix form as:

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{P} \quad (2)$$

In this matrix representation:

- \mathbf{X} is the vector of NTT coefficients.
- \mathbf{W} is the NTT matrix, where each element is derived from the powers of ω .
- \mathbf{P} is the vector of polynomial coefficients.

To further explain the matrix form calculation, the NTT matrix \mathbf{W} is constructed by computing the powers of the primitive root ω . Specifically, each element $W_{i,j}$ of \mathbf{W} is defined as $\omega^{ij} \bmod q$, where ω is a primitive n -th root of unity, and q is the modulus (typically a prime number in cryptographic applications). The matrix multiplication $\mathbf{X} = \mathbf{W} \cdot \mathbf{P}$ transforms the vector of polynomial coefficients \mathbf{P} into the NTT domain, represented by the vector \mathbf{X} . This matrix form of the NTT enables efficient computation, especially when leveraging parallel processing in hardware implementations.

NTT plays a crucial role in efficient polynomial multiplication, which is a fundamental operation in many cryptographic algorithms. In particular, the HAETAE, a cryptographic algorithm selected for post-quantum security, utilizes NTT to perform polynomial multiplications efficiently. This capability significantly enhances its performance and security features.

Overall, the NTT's ability to transform polynomial coefficients into a frequency domain representation, execute multiplications, and then revert via the Inverse NTT (Inv_NTT) dramatically reduces the computational complexity compared to traditional polynomial multiplication methods. This reduction in complexity is particularly beneficial for cryptographic algorithms like the HAETAE, where numerous polynomial operations are integral to its functionality.

In our implementation of NTT on ARMv8 processors, we leverage the 128-bit vector registers provided by NEON instructions. ARMv8 processors feature NEON, which allows

for parallel processing of multiple data elements in a single instruction. However, the **movi** instruction, used to load values into vector registers, has a fixed range. To accommodate this, we must efficiently manage the modular Q value and its inverse within the constraints of the 128-bit registers. Specifically, we use four 32-bit values with identical data packed into a single 128-bit vector register. This packing allows for efficient parallel computation and reduces the overhead associated with handling individual values.

Algorithm 1 outlines the polynomial pointwise Montgomery operation. Lines 1–4 load four 32-bit modular Q values into the 128-bit register (**v3**). Lines 5–14 load four 32-bit modular invsere_ Q values into the 128-bit register (**v4**). Lines 15–26 perform the polynomial pointwise Montgomery operation. By taking advantage of the capability to load four 32-bit values into the 128-bit vector register, the number of multiplication operations required, which would otherwise be 256, is reduced to 64.

Algorithm 1 Element-Wise Modular Polynomial Multiplication Utilizing NEON Instruction; (x0: Result of the multiplication operation, x1, x2: Inputs for the multiplication operation)

```

// mk_Q
1: MOVI.4s v3, #0xfc
2: REV16 v3.16b, v3.16b
3: MOVI.4s v5, #0x01
4: ORR.16b v3, v3, v5

// mk_Q_Inv
5: MOVI.4s v4, #0x38
6: MOVI.4s v5, #0x0f
7: REV32 v4.16b, v4.16b
8: SHL.4s v5, v5, #16
9: ORR.16b v4, v4, v5
10: MOVI.4s v5, #0x04
11: REV16 v5.16b, v5.16b
12: ORR.16b v4, v4, v5
13: MOVI.4s v5, #0x01
14: ORR.16b v4, v4, v5

15: MOV x25, #64
16: loop_j:
17: LD1 {v1.4s}, [x1], #16
18: LD1 {v2.4s}, [x2], #16

// mont_reduce
19: SQDMULH v6.4s, v0.4s, v2.4s
20: MUL.4s v27, v2, v4
21: MUL.4s v7, v0, v27
22: SQDMULH v16.4s, v7.4s, v3.4s
23: SHSUB.4s v6, v6, v16

24: ST1 v6.4s, [x0], #16
25: ADD x25, x25, #-1
26: CBNZ x25, loop_j

```

Lines 19–23 implement the Montgomery reduction technique proposed in [21]. The process begins with the use of the `sqdmulh` instruction, which performs signed multiplication of two 32-bit integers a and b , then adds twice the result of this multiplication to itself, and finally returns only the high-order bits of the result, which helps in efficiently managing the results of large multiplications within a limited bit-width. Following this, the `shsub` instruction is used, which subtracts one value (b) from another (a) and then divides the result by two, effectively providing a shifted result that simplifies subsequent operations. In this reduction process, the intermediate result obtained from `sqdmulh` is denoted as c . The result of the modular reduction on b is stored in a temporary register. This temporary value is then multiplied by a , producing a new value referred to as $a1$. Subsequently, $a1$ is combined with the modular inverse of Q in the register using the `sqdmulh` instruction again. The final value is computed by performing additional operations with c and using the `shsub` instruction, which refines the result to ensure correctness in the Montgomery reduction step.

The Montgomery reduction operation in Algorithm 1 is implemented using the techniques from both [19,21]. Lines 19–23 in Algorithm 1 specifically reflect the techniques from [21]. Additionally, our paper includes results implemented using the techniques from both [19,21].

3.1.1. Optimized Implementation of NTT

Algorithm 2 is part of the NTT multiplication operation. The input values $a[j]$ and $a[j+\text{len}]$ are loaded. $a[j]$ is loaded into $v1$ and $a[j+\text{len}]$ is loaded into $v0$. In this process, the $a[j]$ value and $a[j+\text{len}]$ value are loaded from one address value ($x0$), so the address value is directly moved to the desired location and the value is loaded into the register. Afterwards, a multiplication operation is performed, and although the n value of the ring dimension is 256, since parallel implementation is performed, the multiplication operation is performed with 32 operations.

Algorithm 2 Part of NTT utilizing NEON instruction; ($x0$: Result of the multiplication operation, $x1, x2$: Inputs for the multiplication operation)

```
.macro len128
1: MOV x15, #32
2: LD1R v2.4s, [x1], #4

3: loop_i128:
4: LD1 v1.4s, [x0]
5: ADD x0, x0, #512
6: LD1 v0.4s, [x0]

    // mont_reduce
7: SQDMULH v6.4s, v0.4s, v2.4s
8: MUL.4s v27, v2, v4

9: MUL.4s v7, v0, v27
10: SQDMULH v16.4s, v7.4s, v3.4s
11: SHSUB.4s v6, v6, v16

12: SUB.4s v0, v1, v6
13: ST1 v0.4s, [x0]
14: ADD x0, x0, #-512
15: ADD.4s v1, v1, v6
16: ST1 v1.4s, [x0], #16

17: ADD x15, x15, #-1
18: CBNZ x15, loop_i128
.endm
```

3.1.2. Data Reordering as a Permutation

To implement the NTT and inverse NTT operations in parallel, masking must be applied to the 32-bit values used in the multiplication operation. Masking is achieved through a data reordering technique. In order to optimize the parallel implementation of operations using the same zeta value, data reordering was used for some operations. Figure 2 illustrates part of this data reordering process. The top two data values represent the state before register realignment, while the bottom two images show the state after register realignment. The task of rearranging the upper two register arrays to match the lower two arrays was efficiently accomplished using the ZIP1, ZIP2, and TRN1 instructions.

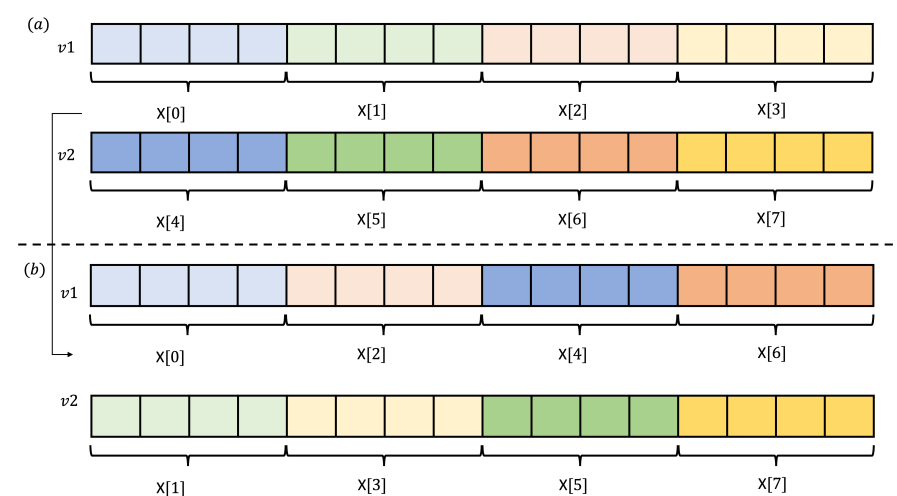


Figure 2. Data Reordering using Neon instruction. The upper section (a) shows the data before reordering, while the lower section (b) displays the data after reordering. ($v1$ and $v2$ is vector register).

3.2. Optimized Implementation of Inverse NTT

Pre-Computation

The zeta value used in the inverse NTT operation is derived by repeatedly applying the inverse NTT operation to the zeta value used in the NTT operation. Since the zeta value required for the inverse NTT operation is a value that inverts the sign of the value located at the end of the zetas array, this value can be precomputed. Therefore, this paper proposes precomputing the zeta value for the inverse NTT and then using the precomputed value during the inverse NTT calculation.

Apart from this, the inverse NTT operation was implemented similarly to the NTT operation. Algorithm 3 illustrates part of the inverse NTT operation.

Algorithm 3 Part of Inverse NTT utilizing NEON instruction; (x0: Result of Multiplication Operation, x1, x2: Input of Multiplication Operation)

```
.macro len1
1: MOV x13, #32

2: loop:
3: LD1 {v2.4s}, [x1], #16
4: MUL.4s v27, v2, v4
5: LD1 {v1.2s}, [x0], #8
6: LD1 {v0.2s}, [x0], #8

    // data_reordering
7: TRN1.4s v17, v1, v0
8: TRN2.4s v18, v1, v0
9: LD1 {v1.2s}, [x0], #8
10: LD1 {v0.2s}, [x0]
11: TRN1.2s v12, v1, v0
12: TRN2.2s v19, v1, v0
13: ZIP1.2d v8, v17, v12
14: ZIP1.2d v9, v18, v19

15: MOV.4s v6, v8
16: ADD.4s v8, v9, v6
17: SUB.4s v9, v6, v9

    // mont_reduce
18: SQDMULH v6.4s, v9.4s, v2.4s
19: MUL.4s v7, v9, v27
20: SQDMULH v16.4s, v7.4s, v3.4s
21: SHSUB.4s v6, v6, v16

22: ZIP1.4s v0, v8, v6
23: ZIP2.4s v1, v8, v6

24: ADD x0, x0, #-24
25: ST1 {v0.4s}, [x0], #16
26: ST1 {v1.4s}, [x0], #16
27: ADD x13, x13, #-1
28: CBNZ x13, loop
.endm
```

4. Evaluation

In this section, we present the performance evaluation of the proposed implementation compared to the reference C implementation [23]. The implementations were developed using Xcode 15.4 and executed within the Xcode IDE. The evaluations were performed on a 2021 MacBook Pro 16" equipped with an Apple M1 Pro chip, operating at up to 3.2 GHz. To achieve optimal performance, the code was compiled using the -O3 optimization flag, which ensures maximum speed.

For performance benchmarking, the optimized HAETAE algorithm was tested by recording the execution time (in milliseconds) over 10,000 iterations. Additionally, we measured the time (in milliseconds) for 1,000,000 iterations to evaluate the performance of the state-of-the-art NTT implementation techniques, including the optimizations proposed in this paper.

The performance measurement results of the high-speed HAETAE algorithm are shown in Table 4, and the performance measurement results for the multiplication implementation using state-of-the-art techniques are presented in Table 5.

Table 4. Performance evaluation result of HAETAE algorithm (unit: ms); notation (B): best performance.

Scheme	Cheon et al. [23]			This Work			This Work (B)		
	Keygen	Sign	Verify	Keygen	Sign	Verify	Keygen	Sign	Verify
HAETAE120	1288	7616	400	1126	6869	322	1114	6818	316
HAETAE180	1502	4198	694	1303	3749	554	1295	3721	545
HAETAE260	2407	76,081	862	2173	67,758	710	2177	66,864	692

Table 5. Performance evaluation results of the NTT algorithm implementation using the latest implementation techniques (unit: ms); notation (B): best performance.

	Cheon et al. [23]	This Work	This Work (B)
NTT	900	319	293
Inverse NTT	1157	373	319
Poly pointwise Montgomery	247	58	27

The performance results of applying the masking technique proposed in this paper, combined with state-of-the-art NTT implementation techniques that incorporate pre-computation, are detailed below.

Initially, employing the implementation technique outlined in [19], we observed significant performance improvements. Specifically, this optimized approach achieved enhancements of $2.82\times$ for the NTT operation, $3.10\times$ for the Inverse-NTT operation, and $4.72\times$ for pointwise Montgomery multiplication.

Further gains were achieved using an alternative implementation technique referenced in [21]. This method demonstrated even more substantial performance improvements, with enhancements of $3.07\times$ for NTT, $3.63\times$ for Inverse-NTT, and a remarkable $9.15\times$ for pointwise Montgomery operations.

When these advanced techniques were applied to the HAETAE encryption algorithm, the performance metrics were as follows: For HAETAE120, the key generation (Keygen) operation saw a performance improvement of $1.16\times$, the signature (Sign) operation improved by $1.12\times$, and the verification (Verify) operation realized a $1.27\times$ boost. Similarly, for HAETAE180, the Keygen operation improved by $1.16\times$, the Sign operation by $1.13\times$, and the Verify operation by $1.27\times$. Finally, for HAETAE260, the Keygen operation demonstrated a performance increase of $1.11\times$, the Sign operation improved by $1.14\times$, and the Verify operation exhibited a performance enhancement of $1.25\times$.

5. Conclusions

In this paper, we proposed optimized implementations of the HAETAE digital signature algorithm for ARMv8 processors and evaluated their performance on Apple M1 Pro processors. Our approach leverages NEON instructions inherent to the ARMv8 architecture, allowing us to capitalize on its parallel processing capabilities. We applied masking and pre-computation techniques to state-of-the-art NTT implementation methods, achieving substantial performance improvements.

Especially, our optimized implementations achieved performance improvements of up to $3.07\times$ for the NTT operation, up to $3.63\times$ for Inverse-NTT, and up to $9.15\times$ for pointwise Montgomery multiplication. These advancements translate directly into the performance metrics of the HAETAE algorithm. Specifically, the key generation algorithm demonstrated a performance improvement of up to $1.16\times$, the signing algorithm saw an enhancement of up to $1.14\times$, and the verification algorithm exhibited an impressive improvement of up to $1.25\times$.

These findings highlight the efficacy of the proposed techniques in improving HAETAE's performance on ARMv8 processors. The significant improvements in NTT and Montgomery operations contribute directly to the overall efficiency of the HAETAE algorithm, making it more appropriate for use in resource-limited environments.

In future research, we suggest investigating additional optimizations for NTT multiplication on ARMv8 processors, aiming to surpass the performance of existing methods. Moreover, the optimizations presented in this work could potentially be extended to other cryptographic primitives, such as digital signatures and hash functions.

Author Contributions: Software, M.S. and M.L.; Writing—original draft, M.S.; Writing—review & editing, H.S.; Supervision, H.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00264, Research on Blockchain Security Technology for IoT Services, 50%) and this work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00627, Development of Lightweight BIoT technology for Highly Constrained Devices, 50%).

Data Availability Statement: The data supporting the reported results are available at https://github.com/minjoo97/HAETAE_on_ARMv8.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Feynman, R.P. Simulating physics with computers. In *Feynman and Computation*; CRC Press: Boca Raton, FL, USA, 2018; pp. 133–153.
2. Shor, P.W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.* **1999**, *41*, 303–332. [\[CrossRef\]](#)
3. Grover, L.K. A fast quantum mechanical algorithm for database search. In Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, Philadelphia, PA, USA, 22–24 May 1996; pp. 212–219.
4. Choi, C.Q. IBM’s Quantum Leap: The Company Will Take Quantum Tech Past the 1,000-Qubit Mark in 2023. *IEEE Spectr.* **2023**, *60*, 46–47. [\[CrossRef\]](#)
5. Yan, B.; Tan, Z.; Wei, S.; Jiang, H.; Wang, W.; Wang, H.; Luo, L.; Duan, Q.; Liu, Y.; Shi, W.; et al. Factoring integers with sublinear resources on a superconducting quantum processor. *arXiv* **2022**, arXiv:2212.12372.
6. Hossain, M.; Kayas, G.; Hasan, R.; Skjellum, A.; Noor, S.; Islam, S.R. A Holistic Analysis of Internet of Things (IoT) Security: Principles, Practices, and New Perspectives. *Future Internet* **2024**, *16*, 40. [\[CrossRef\]](#)
7. Kumar, A.; Ottaviani, C.; Gill, S.S.; Buyya, R. Securing the future internet of things with post-quantum cryptography. *Secur. Priv.* **2022**, *5*, e200. [\[CrossRef\]](#)
8. Balogh, S.; Gallo, O.; Ploszek, R.; Špaček, P.; Zajac, P. IoT security challenges: Cloud and blockchain, postquantum cryptography, and evolutionary techniques. *Electronics* **2021**, *10*, 2647. [\[CrossRef\]](#)
9. Kumari, S.; Singh, M.; Singh, R.; Tewari, H. Post-quantum cryptography techniques for secure communication in resource-constrained Internet of Things devices: A comprehensive survey. *Softw. Pract. Exp.* **2022**, *52*, 2047–2076. [\[CrossRef\]](#)
10. Shamshad, S.; Riaz, F.; Riaz, R.; Rizvi, S.S.; Abdulla, S. An enhanced architecture to resolve public-key cryptographic issues in the internet of things (IoT), employing quantum computing supremacy. *Sensors* **2022**, *22*, 8151. [\[CrossRef\]](#)
11. Malina, L.; Popelova, L.; Dzurenda, P.; Hajny, J.; Martinasek, Z. On feasibility of post-quantum cryptography on small devices. *IFAC-PapersOnLine* **2018**, *51*, 462–467. [\[CrossRef\]](#)
12. NIST PQC Project. Available online: <https://csrc.nist.gov/Projects/post-quantum-cryptography> (accessed on 21 July 2024).
13. KpqC Competition. Available online: <https://kpqc.or.kr/competition.html> (accessed on 21 July 2024).
14. Oder, T.; Speith, J.; Hölting, K.; Güneysu, T. Towards practical microcontroller implementation of the signature scheme Falcon. In *Proceedings of the Post-Quantum Cryptography: 10th International Conference, PQCrypto 2019, Chongqing, China, 8–10 May 2019*; Revised Selected Papers 10; Springer: Berlin/Heidelberg, Germany, 2019; pp. 65–80.
15. Chen, M.S.; Chou, T. Classic McEliece on the ARM cortex-M4. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*; IACR: Lyon, France, 2021; pp. 125–148.
16. Sim, M.; Eum, S.; Kwon, H.; Kim, H.; Seo, H. Optimized implementation of encapsulation and decapsulation of Classic McEliece on ARMv8. *Cryptol. ePrint Arch.* **2022**, 2022/1706. Available online: <https://eprint.iacr.org/2022/1706> (accessed on 23 September 2024).
17. Nguyen, D.T.; Gaj, K. Fast falcon signature generation and verification using armv8 neon instructions. In *Proceedings of the International Conference on Cryptology in Africa*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 417–441.
18. Huang, J.; Adomnicăi, A.; Zhang, J.; Dai, W.; Liu, Y.; Cheung, R.C.; Koç, Ç.K.; Chen, D. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2024**, *2024*, 1–24. [\[CrossRef\]](#)
19. Kim, Y.; Song, J.; Youn, T.Y.; Seo, S.C. Crystals-Dilithium on ARMv8. *Secur. Commun. Netw.* **2022**, *2022*, 5226390. [\[CrossRef\]](#)

20. Seo, S.C.; An, S. Parallel implementation of CRYSTALS-Dilithium for effective signing and verification in autonomous driving environment. *ICT Express* **2023**, *9*, 100–105. [CrossRef]
21. Becker, H.; Hwang, V.; Kannwischer, M.J.; Yang, B.Y.; Yang, S.Y. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *Cryptol. ePrint Arch.* **2021**, 2021/986. Available online: <https://eprint.iacr.org/2021/986> (accessed on 23 September 2024). [CrossRef]
22. Seo, H.; Sanal, P.; Jalali, A.; Azarderakhsh, R. Optimized implementation of SIKE round 2 on 64-bit ARM Cortex-A processors. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 2659–2671. [CrossRef]
23. Cheon, J.H.; Choe, H.; Devevey, J.; Güneysu, T.; Hong, D.; Krausz, M.; Land, G.; Möller, M.; Stehlé, D.; Yi, M. Haetae: Shorter lattice-based fiat-shamir signatures. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2024**, *2024*, 25–75. [CrossRef]
24. Kwon, H.; Sim, M.; Song, G.; Lee, M.; Seo, H. Evaluating kqc algorithm submissions: Balanced and clean benchmarking approach. In *Proceedings of the International Conference on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 338–348.
25. Cottaar, J.; Hövelmanns, K.; Hülsing, A.; Lange, T.; Mahzoun, M.; Pellegrini, A.; Ravagnani, A.; Schäge, S.; Trimoska, M.; de Weger, B. Report on evaluation of KpqC candidates. *Cryptol. ePrint Arch.* **2023**, 2023/1853. Available online: <https://eprint.iacr.org/2023/1853> (accessed on 23 September 2024).
26. Choi, Y.; Kim, M.; Kim, Y.; Song, J.; Jin, J.; Kim, H.; Seo, S.C. KpqBench: Performance and Implementation Security Analysis of KpqC Competition Round 1 Candidates. *IEEE Access* **2024**. [CrossRef]
27. Lee, J.; Lee, E.m.; Kim, J. Security Analysis on TiGER KEM in KpqC Round 1 Competition Using Meet-LWE Attack. *J. Korea Inst. Inf. Secur. Cryptol.* **2023**, *33*, 709–719.
28. Ikematsu, Y.; Jo, H.; Yasuda, T. A security analysis on MQ-Sign. In *Proceedings of the International Conference on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 40–51.
29. Kim, S.; Lee, E.M.; Lee, J.; Lee, M.J.; Noh, H. Security Evaluation on KpqC Round 1 Lattice-Based Algorithms Using Lattice Estimator. In *Proceedings of the International Conference on Information Security and Cryptology*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 261–281.
30. NIST PQC Project: Digital Signature Schemes. Available online: <https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures> (accessed on 21 July 2024).
31. Lyubashevsky, V. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 598–616.
32. Lyubashevsky, V. Lattice signatures without trapdoors. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 738–755.
33. Devevey, J.; Fawzi, O.; Passelègue, A.; Stehlé, D. On rejection sampling in lyubashevsky's signature scheme. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 34–64.
34. Abdulrahman, A.; Hwang, V.; Kannwischer, M.J.; Sprenkels, A. Faster kyber and dilithium on the cortex-m4. In *Proceedings of the International Conference on Applied Cryptography and Network Security*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 853–871.
35. Armv8-A Instruction Set Architecture. Available online: <https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets> (accessed on 21 July 2024).
36. Kwon, H.; Kim, H.; Sim, M.; Eum, S.; Lee, M.; Lee, W.K.; Seo, H. ARMing-Sword: Scabbard on ARM. In *Proceedings of the International Conference on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 237–250.
37. Kwon, H.; Kim, H.; Sim, M.; Lee, W.K.; Seo, H. Look-up the Rainbow: Table-based Implementation of Rainbow Signature on 64-bit ARMv8 Processors. *ACM Trans. Embed. Comput. Syst.* **2023**, *22*, 80. [CrossRef]
38. Sim, M.; Kwon, H.; Eum, S.; Song, G.; Lee, M.; Seo, H. Efficient Implementation of the Classic McEliece on ARMv8 Processors. In *Proceedings of the International Conference on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2023; pp. 324–337.
39. Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schwabe, P.; Seiler, G.; Stehlé, D. Crystals-Dilithium: A lattice-based digital signature scheme. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*; IACR: Lyon, France, 2018; pp. 238–268.
40. Montgomery, P.L. Modular multiplication without trial division. *Math. Comput.* **1985**, *44*, 519–521. [CrossRef]
41. Chung, C.M.M.; Hwang, V.; Kannwischer, M.J.; Seiler, G.; Shih, C.J.; Yang, B.Y. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*; IACR: Lyon, France, 2021; pp. 159–188.
42. Zhang, N.; Yang, B.; Chen, C.; Yin, S.; Wei, S.; Liu, L. Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*; IACR: Lyon, France, 2020; pp. 49–72.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.