applied sciences

Article

Optimizing HAWK Signature Scheme Performance on ARMv8

Siwoo Eum [1], Minwoo Lee [2] and Hwajeong Seo [2,*]

1 Department of Information Computer Engineering, Hansung University, Seoul 02876, Republic of Korea
2 Department of Convergence Security, Hansung University, Seoul 02876, Republic of Korea
* Correspondence: hwajeong@hansung.ac.kr

Abstract: This study proposes an optimized implementation of the HAWK Signature algorithm, one of the candidates in the first evaluation round for additional digital signature schemes in the NIST Post-Quantum Cryptography competition. The core motivation of this research is to improve the performance of HAWK algorithm. By conducting profiling analysis to identify, we identified the most resource-intensive functions. And then we optimized the functions. The optimization techniques through profiling analysis are not limited to HAWK but can be applied to other algorithms as well. Additionally, the study demonstrates how efficient optimization can be achieved using fewer instructions by leveraging lesser-known ARMv8 instructions. By targeting the functions with the highest overhead and utilizing fewer instructions, a performance improvement of approximately 2.5% for Hawk512 and 4% for Hawk1024 was achieved, respectively. These results confirm that combining profiling analysis with efficient instruction usage can lead to significant performance improvements.

Keywords: HAWK; post-quantum cryptography; digital signature; optimized implementation; ARMv8; profiling

Citation: Eum, S.; Lee, M.; Seo, H. Optimizing HAWK Signature Scheme Performance on ARMv8. *Appl. Sci.* **2024**, *14*, 8647. https://doi.org/10.3390/app14198647

Academic Editors: Ian D. Wilson and Reza Montasari

Received: 4 September 2024
Revised: 20 September 2024
Accepted: 23 September 2024
Published: 25 September 2024

1. Introduction

NIST is actively working on the development of quantum-resistant cryptography to protect current cryptographic systems that are threatened by the advent of quantum computers. Currently, four final algorithms have been selected through the NIST PQC and standardization is in progress [1–3]. Among the final selected algorithms are three digital signature algorithms: CRYSTALS-Dilithium, FALCON and SPHINCS+ [4–6]. However, to select additional digital signature algorithms, NIST requested additional digital signature candidates in 2022, and 40 algorithms are currently in the first round [7].

The HAWK algorithm is currently one of the first-round candidates for additional digital signature algorithms. NIST did not initially plan to accept lattice-based algorithms, but allowed the submission of algorithms with better performance than the final algorithm selected in the PQC competition, making the first round possible.

Quantum-resistant cryptography is designed with more complex mathematical theories to provide higher security than modern cryptography. These more complex mathematical theories require more computation than modern cryptography, resulting in longer computation times and higher memory requirements. To address these issues, extensive research is being conducted on the memory and performance of newly developed quantum-resistant encryption algorithms [8–10].

In [8], during the signature-generation process on ARMv8 processors, a vectorized scalable FFT implementation applicable to FFT levels greater than 5 was developed. In this process, the twiddle factor table was compressed using the complex conjugate roots of FFT. Specifically, the table size was reduced from 16KB in the reference implementation to 4KB. The modified FFT implementation, utilizing a twiddle factor table that is 4 times smaller, is a universal technique that is not limited to a specific processor. Therefore, we propose an efficient implementation technique that can be applied to various environments, including platforms with limited memory or storage capacity.

Ref. [9] proposed an optimized implementation of the encapsulation and decapsulation processes on ARMv8. In the encapsulation process, the performance was improved by optimizing the computations during syndrome generation, utilizing the fact that most identity matrices are zero to omit unnecessary calculations. In the decapsulation process, the study optimized the multiplication and inversion operations, which consume a significant amount of time in the extended binary finite field F_{2^m} (where $m = 12$ or 13), by efficiently implementing these operations using ARM instructions.

Through studies like these, it has been confirmed that optimizations on ARMv8 can enable more efficient execution of complex algorithms such as cryptographic computations and mathematical operations. However, for high-performance processors like ARMv8, improper optimization methods can lead to performance degradation. To address these challenges, it is essential to leverage the characteristics of the ARMv8 architecture and improve computational efficiency by utilizing ARMv8-specific instructions. This study aims to optimize the key-generation process of the HAWK scheme on ARMv8 to enhance performance. We propose an optimization technique that uses efficient instructions based on detailed profiling and algorithmic analysis of the computation process. The proposed optimization will contribute to the broader use of ARMv8 in security-sensitive applications. Our main contributions are as follows.

Contributions

1. We present an optimized implementation by profiling and analyzing the computational process of the HAWK Signature scheme. Through the profiling process, we discovered the overhead occurring during the computation process and optimized it to reduce the overhead. Profiling analysis not only helps in optimization implementation, but also provides higher confidence in the results of this paper by presenting the analysis results. This profiling analysis method can help deepen the understanding of the HAWK algorithm. Additionally, by proposing an analysis method that is not limited to the HAWK algorithm but applicable to any other algorithm, it introduces an analysis method that can be applied to various algorithms.
2. We implemented a parallel version of the NTT, which is frequently used in lattice algorithms, to optimize the HAWK algorithm. However, we found that at lower $\log n$ values, the additional steps required for parallel implementation led to performance degradation. By analyzing the performance based on $\log n$, we were able to verify the reliability of the Parallel NTT implementation. As a result, we identified the $\log n$ values where performance improvements occur. Consequently, we optimized the algorithm by using the Reference NTT for lower $\log n$ values and applying the parallel implementation for higher $\log n$ values.
3. We present an optimized implementation of the HAWK Signature scheme on ARMv8. This optimization, based on a precise analysis of the algorithm's operations, leverages relevant instructions provided by the ARMv8 architecture, such as 'sbfx' and 'csel', to achieve the same computational results with fewer instructions. These instructions contribute to reducing execution time by efficiently processing data and handling conditional operations without branching. Additionally, by avoiding branches, we ensure security through constant-time implementation.

2. Background

2.1. HAWK Signature Scheme [11]

The HAWK algorithm is a signature algorithm based on the lattice isomorphism problem (LIP) [12]. The lattice isomorphism problem is based on the problem of determining whether two lattice structures are identical using a mathematical structure called a lattice. HAWK has a similar name to FALCON. This reason is not very similar to the FALCON Signature method, but the NTRU [13] equation is solved with similar settings during the key-generation process and the FALCON code is reused when implementing HAWK, so it was named similarly. HAWK designed its algorithm with an emphasis on simplicity. Due to this simplicity, it was designed to be free of floating point and rejection sampling. This simple design results in faster signature speeds and smaller signature sizes when compared to FALCON.

HAWK provides two parameter sets, HAWK-512 and HAWK-1024, and satisfies the security levels of NIST-1 and NIST-5, respectively. The public key size is larger than that of FALCON, a similar scheme, but the private key and signature sizes are smaller. Detailed key size and signature size are listed in Table 1.

Table 1. Key and signature size for HAWK in bytes.

	Private Key	Public Key	Signature
HAWK-512	184	1024	555
HAWK-1024	360	2440	1221
FALCON-512	1281	897	666
FALCON-1024	2305	1793	1280

2.1.1. HawkKeyGen (Algorithm 1)

Polynomials f and g are generated by independently sampling from the central binomial distribution according to specific conditions. Create F and G that satisfy NTRUSolve through the generated f and g. NTRUSolve is the problem of finding polynomials F and G that satisfy $fG \times Gf = 1$. Create a matrix B using the generated f, F, g and G polynomials, and generate the public key Q through B. The private key is generated through the generated B matrix and the hash value of the public key.

Algorithm 1 HawkKeyGen.

```
Ensure: $B \in GL_2(R_n)$ and $Q = B^*B$
1: Sample coefficients of $f, g \in R_n$ i.i.d. from Bin($\eta$)
2: if $f$-$g$-conditions$(f, g)$ is false then restart
3: $r \leftarrow$ NTRUSolve$(f, g)$
4: if $r$ is $\perp$ then restart
5: $(F, G) \leftarrow r$
6: $B \leftarrow \begin{pmatrix} f & F \\ g & G \end{pmatrix}$, $Q \leftarrow B^*B$
7: if KGen-encoding$(Q, B)$ is false then restart
8: $h_{\text{pub}} \leftarrow H(Q)$
9: return $(pk, sk) \leftarrow (Q, (B, h_{\text{pub}}))$
```

2.1.2. HawkSign (Algorithm 2)

A hash M is generated through the message (m) and the private key hash (hpub), and a hash value h is generated by combining M and salt. Sample vector x from the expanded lattice using the generated h. Compute the vector w using x, and generate the signature vector s using w and h. The generated s is compressed to create a signature value s1, and the final signature is created by combining Salt and s1.

Algorithm 2 HawkSign.

```
Require: A message $\mathbf{m}$ and secret key $sk = (B, h_{\text{pub}})$
Ensure: A signature sig formed of a uniform salt $s \in \{0,1\}^{\text{saltlen}}$ and $s_1 \in R_n$
1: $\mathbf{M} \leftarrow H(m \| h_{\text{pub}})$
2: $\mathbf{salt} \leftarrow \text{Rnd}(\text{saltlen}_{bits})$
3: $\mathbf{h} \leftarrow H(M \| salt)$
4: $\mathbf{t} \leftarrow B \cdot h \mod 2$
5: $\mathbf{x} \leftarrow D_{2\mathbb{Z}^{2n}} + t, 2\sigma_{sign}$
6: if $||x||^2 > 4 \cdot \sigma^2_{\text{verify}} \cdot 2n$ then restart
7: $\mathbf{w} \leftarrow B^{-1}x$
8: if sym-break$(w)$ is false then $\mathbf{w} := -\mathbf{w}$
9: $\mathbf{s} \leftarrow \frac{1}{2}(\mathbf{h} - \mathbf{w})$
10: $s_1 \leftarrow \text{Compress}(s)$
11: if sig-encoding$(salt, s_1)$ is false then restart
12: return $sig \leftarrow (salt, s_1)$
```

2.1.3. HawkVerify (Algorithm 3)

Create h_{pub} using the public key and combine it with the message to create hash M. Extract the salt value from the *sig* and combine M and salt to generate h. Restore s with the calculated h, Q and s_1, and calculate w through the restored s. Check whether the signature value is valid by checking whether the calculated w meets certain conditions.

Algorithm 3 HawkVerify.

```
Require: A message $m$, a public key $pk = Q$ and a signature $sig = (salt, s_1)$
Ensure: A bit determining whether $sig$ is a valid signature on $m$
1: $h_{\text{pub}} \leftarrow H(Q)$
2: $M \leftarrow H(m \| h_{\text{pub}})$
3: $\mathbf{h} \leftarrow H(M \| salt)$
4: $s \leftarrow \text{Decompress}(s_1, \mathbf{h}, \mathbf{Q})$
5: $\mathbf{w} \leftarrow \mathbf{h} - 2s$
6: if $\text{len}_{\text{bits}}(salt) = \text{saltlen}_{\text{bits}}$ and $s \in R_n^2$ and sym-break$(w)$ and $\|w\|^2_Q \leq 4 \cdot \sigma^2_{\text{verify}} \cdot 2n$
   then
7:     return 1
8: else
9:     return 0
```

2.2. Chinese Remainder Theorem [14]

The Chinese Remainder Theorem (CRT) is a fundamental result in number theory that provides a solution to a system of simultaneous congruences with pairwise coprime moduli. The theorem can be stated as follows:

Let $n_1, n_2, \ldots, n_k$ be pairwise coprime positive integers, and let $a_1, a_2, \ldots, a_k$ be integers. Then, there exists an integer x that simultaneously satisfies the system of congruences:

$$\begin{aligned}
x &\equiv a_1 \pmod{n_1}, \\
x &\equiv a_2 \pmod{n_2}, \\
&\vdots \\
x &\equiv a_k \pmod{n_k}.
\end{aligned}$$

The solution x is unique modulo N, where N is the product of all moduli, i.e., $N = n_1 \cdot n_2 \cdot \ldots \cdot n_k$. The solution can be constructed explicitly using the following formula:

$$x \equiv \sum_{i=1}^{k} a_i \cdot N_i \cdot N_i^{-1} \pmod{N},$$

where $N_i = \frac{N}{n_i}$ and N_i^{-1} is the modular multiplicative inverse of N_i modulo n_i, which satisfies the equation:

$$N_i \cdot N_i^{-1} \equiv 1 \pmod{n_i}.$$

This construction ensures that each term $a_i \cdot N_i \cdot N_i^{-1}$ contributes only to the congruence condition for n_i, thus satisfying all the given congruences simultaneously. The Chinese Remainder Theorem is not only a powerful theoretical tool in number theory but also has practical applications in computer science, cryptography and coding theory, where it is used to simplify computations by breaking them into smaller, more manageable parts.

In particular, in the process of finding the polynomials F and G that satisfy the condition $fG - gF = 1$ in the NTRUsolve algorithm (in the HAWK key-generation process), the method of handling polynomial coefficients independently under several small prime moduli and then combining them using the Chinese Remainder Theorem (CRT) simplifies the system's computations. This approach offers the advantage of compensating for computation failures in specific moduli with successful computations in other moduli.

2.3. Number Theoretic Transform

The Number Theoretic Transform (NTT) is a transformation technique based on principles similar to the Fourier Transform, utilizing mathematical properties defined in finite fields and modular arithmetic. It is particularly useful in fields like cryptography and polynomial multiplication, providing efficient computation. NTT operates using modular arithmetic, where integers are reduced modulo a specific prime p. This allows the use of integers instead of complex numbers, which is crucial for performing cryptographically secure operations.

NTT is based on the primitive root of a specific prime p, and the transformation is computed using discrete roots of unity defined in the finite field. Unlike the Fourier Transform that operates over complex numbers, NTT uses mathematical structures in finite fields. This characteristic makes NTT highly efficient for performing fast polynomial multiplication.

Polynomial multiplication plays an important role in post-quantum cryptography (PQC) algorithms, and NTT reduces the computational complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$. As a result, many PQC algorithms are implemented using NTT, and numerous studies have been conducted to optimize the efficient implementation of NTT for enhanced performance. [15–17].

2.4. ARMv8 Architecture

ARMv8 is a 64-bit high-performance architecture that supports both AArch64 and AArch32. ARMv8 provides the flexibility to fulfill a wide range of applications and system requirements through two states. ARMv8 provides 31 general-purpose registers (x0–x30) that can be used in 64 bits, and when you want to use them in 32 bits, you can use w0–w30. Additionally, ARMv8 supports vector registers. It provides thirty-two 128 -bit registers (v0–v31), and parallel operations on multiple data elements are possible through vector registers and SIMD instructions [18]. The detailed description is available in Table 2.

Table 2. Summarized instruction set of AMRv8 for optimized Hawk

Instruction	Description	Operands	Operation
add	Add	Rd, Rn, Rm	Rd = Rn + Rm
and	And	Rd, Rn, Rm	Rd = Rn & Rm
asr	Arithmetic Shift Right	Rd, Rn, #imm	Rd = Rn >>#imm
cbz	Compare and Branch on Zero	Rn, label	if (Rn==0) branch to label
cmp	Compare	Rn, Rm	Set Flags based on Rn - Rm
csel	Conditional Select	Rd, Rn, Rm, cond	Rd = (cond) ? Rn : Rm
lsr	Logical Shift Right	Rd, Rn, #imm	Rd = Rn >>#imm
sbfx	Signed Bit Field Extract	Rd, Rn, #lsb, #width	Rd = SignExtend((Rn>>#lsb) & ((1<<#width) −1))

3. HAWK Optimization Through Profiling

This section describes the results of an analysis profiling the operation process of the reference code of HAWK. In this paper, we present an optimized implementation of HAWK by identifying processes that generate overhead through profiling analysis and optimizing those processes.

3.1. Profiling of the HAWK Signature Scheme

Before implementing the optimization, time profiling was conducted to analyze the computational process that generates high overhead in the HAWK Signature. This time profiling process can be of great help in implementing optimization. If a less-used operation is implemented optimally, there may be performance improvement, but it has little effect on the overall operation process, so it is difficult to expect overall performance improvement. In other words, greater performance improvement can be expected by optimizing operations that are used 10% of the process B than by optimizing operations that are used only 1% of the process.

The Time Profile tool provided by Xcode was used for profiling analysis, and measurements were made on a 13-inch 2020 MacBook Pro equipped with Apple M1. As for the code used for analysis, profiling analysis was performed based on the reference code submitted by the HAWK development team (https://csrc.nist.gov/projects/pqc-dig-sig/round-1-additional-signatures, accessed on 20 September 2024).

In Table 3, When the HAWK algorithm's key-generation, signature-generation and signature-verification processes are viewed as a single process, it shows the process that generates the greatest overhead. As shown in Table 3, the largest overhead occurs during the key-generation process, which accounts for 98% of the entire process. Since the key-generation process takes up most of the HAWK algorithm, optimized implementation of key generation in the HAWK algorithm is essential. Therefore, in order to optimize the key-generation process, a more detailed profiling analysis was performed on the key-generation process.

Table 3. Profiling analysis results for the entire HAWK Signature scheme process (parameter: hawk-1024, loop: 1000).

Symbol Name	Time	Weight
Total	15.54 (s)	100.0%
crypto_sign_keypair	15.30 (s)	**98.4%**
crypto_sign	133.00 (ms)	0.8%
crypto_sign_open	111.00 (ms)	0.7%

3.1.1. Process of Key Generation

The results of analyzing the operations used in the key-generation process are shown in Table 4. The calculation processes that generate the greatest overhead during the key-generation process are the NTRUSolve calculation process and the Rebuild_CRT calculation process. NTRUSolve has a 66.9% weight in the key-generation process, and the Rebuild_CRT process has a 27.9% weight. These two processes account for 95% of the key-generation process. Therefore, if it can be analyze the operations that are frequently used inside the two operation processes and optimize the operations, it can be improve the performance of the key-generation process of the HAWK Signature.

Table 4. Profiling analysis results for the crypto_sign_keypair process (parameter: hawk-1024, loop: 1000).

Symbol Name	Time	Weight
Total(crypto_sign_keypair)	15.30 (s)	100.0%
hawk_ntrugen_solve_NTRU	10.24 (s)	**66.9%**
hawk_ntrugen_rebuild_CRT	4.28 (s)	**27.9%**
etc.	0.78 (s)	5.2%

As explained in Algorithm 1, NTRUSolve is a problem of finding a polynomial that satisfies $fG \times Gf = 1$ condition using sampled f and g to generate F and G. Considering the conditions, polynomial multiplication, addition and modular operations can be expected to be commonly used operations in NTRUSolve. CRT is the Chinese remainder theorem. Rebuild_CRT is a method to divide a large number into operations of a smaller number using a series of remainders for coprime modular values, and then restore it back to a large number. For more detailed analysis, the operations used internally in NTRUSolve and Rebuild_CRT were additionally analyzed.

As can be seen in Table 5, the operations mainly used in the two processes are polynomial multiplication and addition, and modular operations. In the HAWK algorithm implementation, the two operations are implemented as $add\_mul\_small()$ and $mod\_small\_unsigned()$ functions. $add\_mul\_small()$ performs the operation of multiplying a polynomial by an integer and adding it with another polynomial, and the $mod\_small\_unsigned()$ function performs a modular operation of dividing the polynomial by small prime numbers and finding the remainder. The two operations($add\_mul\_small(), mod\_small\_unsigned()$), including the NTRUSolve and Rebuild_CRT processes, account for 11% and 42% of the total key-generation process, respectively, and together they account for 53%. Therefore, we analyzed and optimized the implementation code with the goal of optimizing the two operations.

Table 5. Profiling analysis results for the crypto_sign_keypair process in detail (parameter: hawk-1024, loop: 1000).

Symbol Name	Time	Weight
Total(crypto_sign_keypair)	15.30 (s)	100.0%
zint_mod_small_unsigned	6.67 (s)	43.3%
zint_add_mul_small	1.65 (s)	10.6%
etc.	6.98 (s)	46.1%

3.1.2. Processs of Sign

The analysis of the operations used during the Sign process is summarized in Table 6. The computational processes that contribute the most significant overhead during the sign process are the `sig_gauss` and `hawk_ntrugen_hawk_regen_fg` processes. The `sig_gauss` process accounts for 49.2% of the overhead in the Sign process, while the `hawk_ntrugen_hawk_regen_fg` process accounts for 16.6%. The `sig_gauss` function generates values following a Gaussian distribution, and the `hawk_ntrugen_hawk_regen_fg` function regenerates the values of f and g. Both functions use the SHAKE algorithm internally to generate random values. Therefore, it can be concluded that the SHAKE algorithm operations contribute significantly to the overhead in the signing process. Additionally, the NTT operation is also heavily used, accounting for 16.5% of the overhead.

Table 6. Profiling analysis results for the crypto_sign process (parameter: hawk-1024, loop: 1000).

Symbol Name	Time	Weight
Total(crypto_sign)	133.00 (ms)	100.0%
sig_gauss	65.00 (ms)	49.2%
hawk_ntrugen_Hawk_regen_fg	22.00 (ms)	16.6%
mp_NTT	22.00 (ms)	16.5%
etc.	24.00 (ms)	17.7%

3.1.3. Processs of Verify

The analysis of the operations used during the verification process is summarized in Table 7. The computational processes that contribute the most significant overhead during the verification process are the `mp_poly_to_ntt` and `decode_gr` processes. The `mp_poly_to_ntt` process accounts for 27.3% of the overhead in the verification process, while the `decode_gr` process accounts for 20.1%. The `mp_poly_to_ntt` function is responsible for converting polynomials to the NTT domain. The `decode_gr` function is responsible for decompressing data by decoding values from a given buffer using the Golomb-Rice coding method. Additionally, the `mp_poly_to_NTT_autoadj` process accounts for 12.3% of the overhead. As a result, it can be concluded that the NTT conversion process contributes the most significant overhead during the verification process.

Table 7. Profiling analysis results for the crypto_verify process (parameter: hawk-1024, loop: 1000).

Symbol Name	Time	Weight
Total(crypto_verify)	113.00 (ms)	100%
mp_poly_to_NTT	31.00 (ms)	27.3%
decode_gr	23.00 (ms)	20.1%
mp_poly_to_NTT_autoadj	14.00 (ms)	12.3%
etc.	45.00 (ms)	41.3%

3.2. Optimized Implementation of Critical Operations

Optimized implementation methods include minimizing memory access and parallel implementation using vector registers. However, beyond these methods, an important factor in optimization is the use of efficient instructions. In this study, we focus on optimizing computational processes with high overhead, as previously analyzed by profiling in the HAWK algorithm, utilizing the efficient instructions provided by the ARMv8 architecture.

Before explaining the optimization implementation, we will first explain in detail the instruction(csel, sbfx) used to implement the optimization.

First, as shown in Figure 1, the 'csel' instruction stores two different registers(Rn, Rm) into Rd(destination register) depending on the condition. This instruction stores Rn or Rm in Rd depending on the state of the condition flag. Since the value stored varies depending on the condition flag, it is used after executing a instruction that sets a condition flag such as 'cmp' before using 'csel' instruction. There are four condition flags: N(Negative), Z(Zero), C(Carry) and V(Overflow). For the 'lt'(less than) condition used in the figure, it is judged as true if N and V are different.

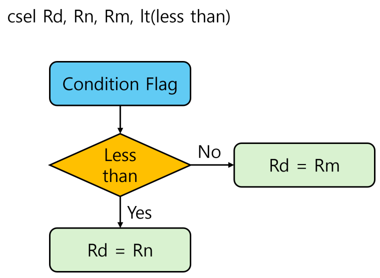

Figure 1. Process of 'csel' instruction operation with less than condition in ARMv8: This diagram illustrates how the 'csel' instruction operates when the condition is 'less than'. Based on the condition flag, if the result is true (less than), *Rd* is set to *Rn*, otherwise it is set to *Rm*.

Next, as shown in Figure 2, The 'sbfx' instruction copies a specific range of bits from a source register to a destination register, while sign-extending the highest bit in the specified range into the higher bits that are not included in the copied range. Here, '#lsb' refers to the starting point of the copied range, and '#width' denotes the length of the range.

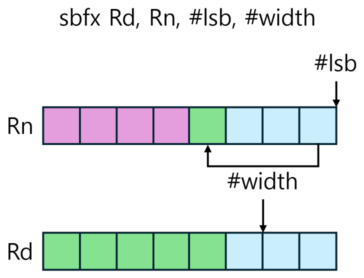

Figure 2. Process of 'sbfx' instruction operation in ARMv8: This diagram illustrates the operation of the 'sbfx' instruction, which extracts a specified number of bits (#width) from a source register (Rn), starting at a given least significant bit position (#lsb), and stores the result in the destination register (Rd).

Through profiling analysis in the previous section, we identified the operations with the highest weight in each computation process. The HAWK algorithm's key-generation process takes the largest portion, with the functions zint_mod_small_unsigned() and zint_add_mul_small() accounting for 43% and 10% of the key-generation process, respectively. We performed optimizations on the operations used in these two functions.

3.2.1. Optimized Implementation of Mod _small _unsigned() Function

Algorithm 4 is a code that implements modular operation to find the remainder by dividing a polynomial by small prime numbers. In line 2 of Algorithm 4, mp_half(R2, p); (Algorithm 5) The computation is performed. This operation performs an operation that divides the value of R2 in half. At this time, if R2 is odd, an operation is performed to divide the value added by the decimal p value in half. In other words, if R2 is an even number, the result is R2/2, and if it is odd, the result is (R2 + p)/2. Distinguishing between even and odd numbers can be done simply by determining whether the least significant bit is 1 or 0. This can be confirmed by 'AND' R2 with 1.

Algorithm 4 Uint32_t zint_mod_small_unsigned.

```
Require: const uint32_t *d, size_t len, size_t stride, uint32_t p, uint32_t
    p0i, uint32_t R2
Ensure: A 32-bit unsigned integer result.
 1: uint32_t x = 0;
 2: uint32_t z = mp_half(R2, p);
 3: d += len * stride;
 4: for uint32_t u = len; u > 0; u- do
 5:     d -= stride;
 6:     uint32_t w = *d - p;
 7:     w += p & tbmask(w);
 8:     x = mp_montymul(x, z, p, p0i);
 9:     x = mp_add(x, w, p);
10: end for
11: return x;
```

Algorithm 5 Uint32_t mp_half.

```
Require: uint32_t *a, uint32_t p
 1: return (a + (p & -(a & 1))) >> 1;
```

Next, there is a method of dividing the operation into 1 and 0 through branching, but using branching is not an efficient method because the length of the code becomes longer and the number of instructions used increases. Next, there is a method of dividing the operation into 1 and 0 through branching, but using branching is not an efficient method because the length of the code becomes longer and the number of instructions used increases. Therefore, for efficient implementation, when R2 is an even number, p is set to 0, and when R2 is an odd number, p is added as is. In other words, it is implemented as R2 + (p&n). At this time, n is set to 0 when it is an even number, and is set to 0xffffffff when it is an odd number. The method of setting n can be simply implemented by adding -1 to the value on which an 'AND' operation was previously performed to check the lowest bit.

In this paper, the 'sbfx' instruction is used for more efficient calculation. The 'sbfx' instruction is an instruction that extracts a specific bit field from a register, expands it and stores it in the register. In other words, the lowest bit can be extracted, expanded and stored. The example code implemented in assembly is as Table 8. As a result, by using the 'sbfx' instruction, the results calculated using the 'and' and 'sub' instruction can be implemented with a single 'sbfx' instruction. Additionally, when branches are used, performance varies depending on whether the number is odd or even. This is not a constant-time implementation. However, if implemented using the 'sbfx' instruction, safety can be guaranteed because the performance is the same regardless of odd or even numbers.

Table 8. Assembly code of uint32_t mp_half()(w0: R2, w1: p, w2: temp).

Before (Using Branch)	After (Using 'sbfx' (Ours))
`and w3, w0, #1`	`sbfx w2, w0, #0, #1`
`cbz w3, _even`	`and w1, w1, w2`
`add w0, w0, w1`	`add w0, w0, w1`
`_even:`	`lsr w0, w0, #1`
`lsr w0, w0, #1`	

Lines 6–7 of Algorithm 4 calculate the w value. In the 6th line, w is declared by subtracting the decimal p value from the *d value, and in the 7th line, the p value is added again. At this time, the p value may be 0 depending on tbmask(w). tbmask(w) returns 0xffffffff if the most significant bit of the w value is 1, and 0 if it is 0. That is, if it is a positive

number, it returns 0, and if it is a negative number, it returns 0xffffffff. When this process is divided into positive and negative cases, the value of w is as follows.

when *d is Positive, $w = *d - p$
when *d is Negative, $w = *d - p + p = *d$

When it is a positive number, the value obtained by subtracting the decimal p value is the result of w. In the case of a negative number, p is added again, so *d is the result. In other words, when it is a negative number, subtracting and adding p is not a necessary operation. Because the first value of w(w = *d − p) can be obtained to know whether it is a positive or negative number, it is difficult to omit the operation of subtracting p. However, in the case of addition operations, it can be omitted if it is known whether the value(w = *d − p) is positive or negative. Therefore, in this paper, we propose an efficient implementation that uses the 'csel' instruction to omit the addition operation and omits tbmask to calculate w with fewer instructions.

Before optimization in Table 9, tbmask(w) is calculated using the 'asr' instruction, and then the 'and' operation is performed with the p value. If w is a positive number, the result of tbmask(w) is stored in the w3 register, and since the most significant bit is 0, 0 is stored in w3. Therefore, in line 4, an addition operation is performed with the w0 register, but since it is 0, the value of the w0 register does not change. On the other hand, if w is a negative number, w3 has the value 0xffffffff in line 2, so w3 stores the p value from the operation result in line 3. Therefore, in line 4, the value of p is added to w0, and the value of p disappears, leaving only *d.

Table 9. Assembly code of computation w(w0: w, w1: *d, w2: p, w3: temp).

Before	After (Using 'csel' (Ours))
1: `sub w0, w1, w2` 2: `asr w3, w0, #31` 3: `and w3, w3, w2` 4: `add w0, w0, w3`	1: `sub w0, w1, w2` 2: `cmp w0, #0` 3: `csel w0, w1, w0, lt`

In the optimized implementation, in order to omit the addition operation, the initial w value (*d − p) is compared with the value of 0 in the second line to check whether it is positive or negative, and when it is negative, through the 'csel' instruction, different values are stored in w0 depending on whether it is negative or positive. The 'csel' instruction stores either w1 or w0 in w0 depending on whether the less than (lt) condition is satisfied based on the result of the comparison with 0 in the second line. If w0 is less than 0 in line 2, the value stored in the w1 register is stored in w0. Conversely, if it is greater than 0, the w0 register value is stored in the w0 register. As a result, the *d value is stored in the w1 register, and the *d − p value is stored in w0, allowing the value to be calculated for both positive and negative numbers without needing to perform the addition instruction.

This process is illustrated in Figures 3 and 4. Figure 3 shows the state before using the 'csel' instruction. As can be seen in the figure, the instruction (represented by the yellow box) is used four times, and depending on the situation, more registers may be required. In contrast, in Figure 4, which shows the use of the 'csel' instruction, the instruction is used three times, and it can be seen that the implementation is possible with just three registers.

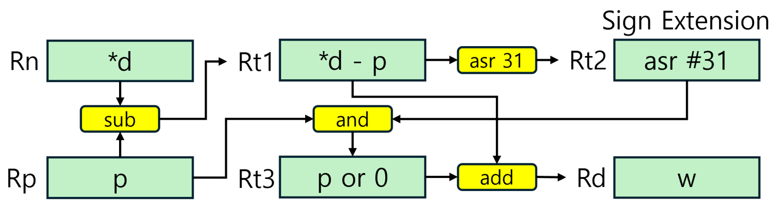

Figure 3. The w calculation process without using the 'csel' instruction: This diagram shows the computation of the value w without using the conditional select instruction. Standard arithmetic operations, such as subtraction, logical and addition, are used to determine the result.

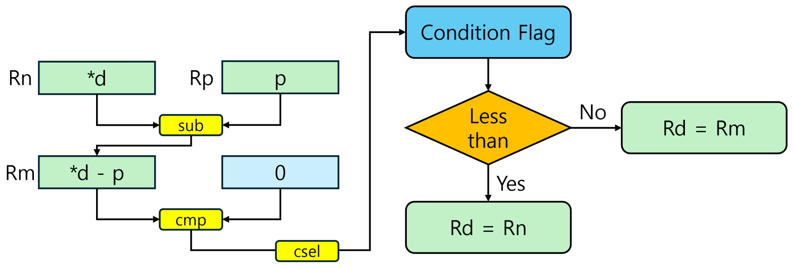

Figure 4. The w calculation process using the 'csel' instruction: This version of the computation uses the 'csel' instruction, which performs a conditional selection based on the comparison of inputs. The process is more efficient compared to the version without 'csel'.

3.2.2. Optimized Implementation of Add _mul _small() Function

In this paper, Algorithm 6 was implemented with fewer instructions than before by using the 'madd' instruction, which can perform multiplication and addition operations with one instruction. The implementation corresponds to line 7 of Algorithm 6. In line 7, to calculate the z value, one multiplication and two addition operations are performed using values such as yw, s, xw and cc. [Implementation 3] is the assembly code for calculating the z value. In the code before optimization, you can see that the process of calculating 'mul' and 'add' is combined into one 'madd' instruction. The detailed description is available in Table 10.

Algorithm 6 Void zint_add_mul_small.

```
Require: uint32_t *restrict x, size_t len, size_t xstride, const uint32_t
    *restrict y, uint32_t s
Ensure: Updated x array.
 1: uint32_t cc = 0;
 2: for size_t u = 0; u < len; u++ do
 3:    uint32_t xw, yw;
 4:    uint64_t z;
 5:    xw = *x;
 6:    yw = y[u];
 7:    z = (uint64_t)yw * (uint64_t)s + (uint64_t)xw + (uint64_t)cc;
 8:    *x = (uint32_t)z & 0x7FFFFFFF;
 9:    cc = (uint32_t)(z ≫ 31);
10:    x += xstride;
11: end for
12: *x = cc;
```

Table 10. Assembly code of computation z(x0: xw, x1: yw, x2: s, x3: cc.)

Before	After (Using 'madd' (Ours))
1: `add x0, x0, x3` 2: `mul x1, x1, x2` 3: `add x0, x0, x1`	1: `add x0, x0, x3` 2: `madd x0, x1, x2, x0`

This process is also illustrated in Figures 5 and 6. As seen in Figure 5, when the 'madd' instruction is not used, the instruction (represented by the yellow box) must be used four times. In contrast, when the 'madd' instruction is used, it can be confirmed that only three instructions are needed, ultimately achieving the same result with fewer instructions by omitting one.

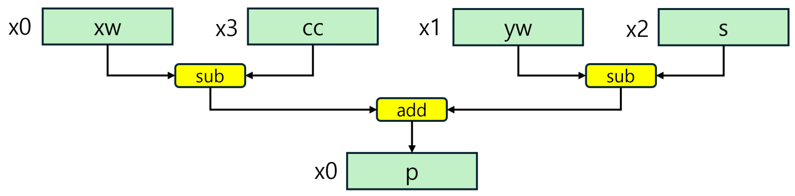

Figure 5. The z computation process without using the 'madd' instruction: This diagram shows the step-by-step computation of the value z using standard arithmetic instructions. Subtraction operations are performed on inputs x0, x3 and x1, x2, followed by an addition to compute the final result.

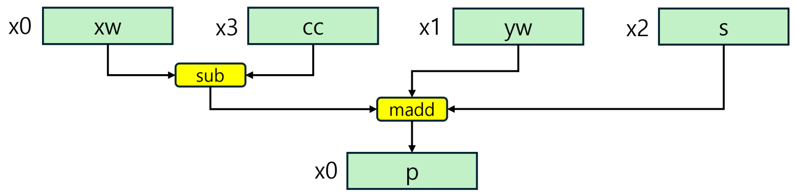

Figure 6. The z computation process using the 'madd' instruction: This version of the computation uses the 'madd' instruction, which combines multiplication and addition in a single step. The process reduces the number of instructions required compared to Figure 5, resulting in a more optimized computation of z.

By utilizing the `sbfx` instruction, bit manipulation operations that would have required multiple instructions can be handled with a single instruction. This reduces clock cycles and improves performance, while also reducing memory access by eliminating the need for separate load/store operations. Additionally, the `csel` instruction allows conditional selection without branching, thereby preventing performance degradation caused by branch misprediction. These instructions optimize execution at the instruction level, minimizing computational overhead and enhancing overall processing performance.

3.2.3. Parallel Implementation of NTT

Our final optimization strategy is the parallel implementation of the Number Theoretic Transform (NTT). In this study, we also attempted an optimized implementation of NTT and proposed a parallel implementation that uses the vector registers of AArch64 for further optimization. Although NTT primarily involves simple operations like multiplication and addition, the high number of repetitions introduces significant overhead. The structure of NTT is depicted in Figure 7.

Figure 7 illustrates the process of NTT (Number Theoretic Transform) computation when $\log n = 3$. As the value of $\log n$ increases, an additional computation stage is added. By arranging the elements that perform identical operations into a single register, parallel implementation becomes possible. In this regard, Figure 7 provides important information.

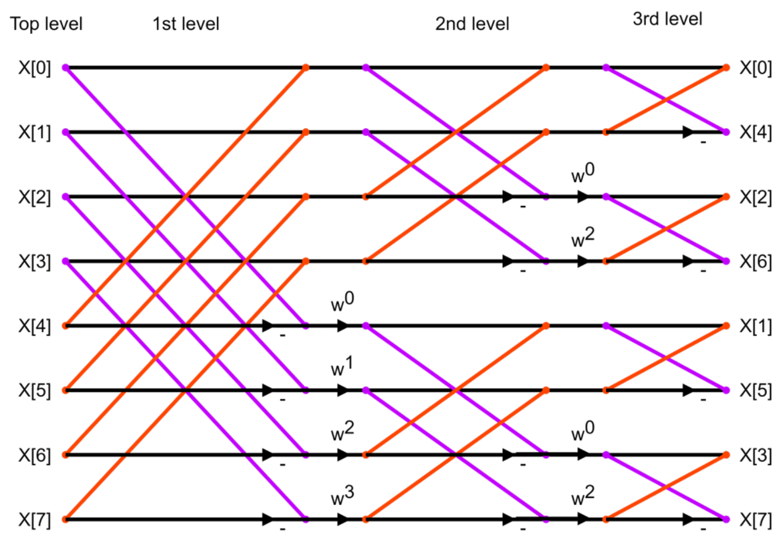

Figure 7. Simple NTT structure (case when logn = 3): This diagram shows an 8-point Number Theoretic Transform (NTT) with three levels, where butterfly operations are performed on pairs of elements using modular multiplication. Coefficients w_0 to w_3 are applied across different levels, illustrating the transformation from input to output.

When examining each stage separately, the parts connected by lines of the same color indicate that they all perform the same operations. Therefore, by aligning the values that perform the same operations into a single register, parallel implementation is achievable. However, since the indices that perform the same operations differ at each stage, different parallel implementation strategies must be applied at each stage. For this reason, it is necessary to check each stage during parallel implementation.

Parallel implementation allows for the simultaneous processing of multiple data, offering better performance compared to serial execution of individual data computations. However, this is not always beneficial. The process of sorting data for parallel computation may introduce overhead, and if this overhead exceeds the performance gain from parallel processing, it can result in a decrease in overall performance. Such cases typically occur when the amount of data to be processed is small. In other words, when there is little computation, simple serial execution can be more efficient than parallel computation.

In the case of NTT, when the value of $\log n$ is small, the number of operations is not large, meaning that serial implementation can be more efficient than parallel implementation. This is because parallel execution may suffer from performance degradation due to the need for sorting data or fetching data from different memory locations. Therefore, in this study, we further analyzed the performance based on different values of $\log n$, in order to identify at which point the performance shift towards parallel implementation occurs.

While parallel implementation can enhance performance by processing multiple data simultaneously, performance degradation may occur due to the need for data alignment, stage verification and, in certain stages, the need to fetch values from different memory locations rather than from contiguous memory addresses. Therefore, the potential performance gains from parallel implementation should be compared against these issues to determine its feasibility.

As shown in Figure 8, up to a $\log n$ value of 7, there is minimal difference in performance between the Reference NTT and the Parallel NTT, with the parallel implementation occasionally exhibiting lower performance. However, beginning from a $\log n$ value of 8, a noticeable performance difference emerges, and this gap continues to widen as $\log n$ increases. This observation suggests that when $\log n$ exceeds 8, parallel implementation can result in performance improvements. In the HAWK Scheme, NTT is employed with $\log n$ values ranging from 1 to 10. Therefore, to optimize performance, parallel implementation should be utilized for specific $\log n$ values, while the Reference NTT should be employed for smaller $\log n$ values. In this context, we determined the specific $\log n$ value to be 8 or higher, based on the reference from Figure 8.

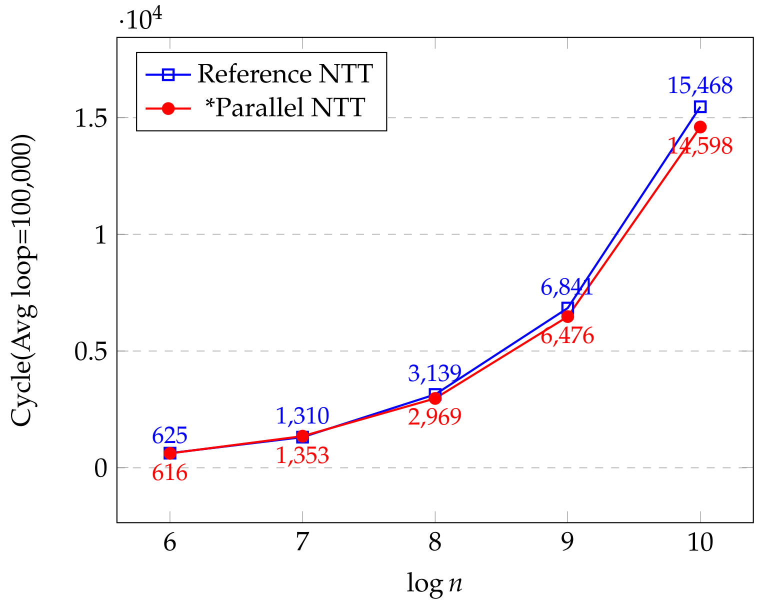

Figure 8. Comparison of Reference and Parallel NTT: This graph compares the performance of the Reference NTT (blue) and the *Parallel NTT (red) based on the average number of cycles over 100,000 loops. As $\log n$ increases, the Parallel NTT consistently shows lower cycle counts compared to the Reference NTT, demonstrating its efficiency, especially at higher values of $\log n$.

4. Evaluation

In this paper, for performance evaluation, performance is measured and evaluated on a 2020 MacBook Pro 13-inch equipped with Apple M1, which is the same environment as the profiling analysis. The 2020 MacBook Pro is equipped with the Apple-designed processor M1, one of the latest ARM processors. The Apple M1 processor is a very high-performance processor with a SoC (System on Chip) that integrates CPU, GPU, DSP and neural network engine into one chip [19].

To compare performance, the operations of the existing implementation and the optimized implementation were compared by calculating the average cycles measured after 1000 repeated operations with the optimization option -O3. The measured results are shown in Table 11.

Table 11. Performance result of HAWK (unit: cycle; loop: 1000).

	KeyPair	**Sign**	**Verify**	**Total**
Hawk512-ref	8,352,475	180,094	158,098	8,690,667
Hawk512-Ours	8,125,339	179,441	157,985	8,462,765
Hawk1024-ref	48,259,761	377,498	329,483	48,966,742
Hawk1024-Ours	46,317,963	377,788	329,533	47,025,284

The performance measurement results for Hawk1024 show identical performance in the signing and verification processes. However, in the key-generation process, which is the focus of this paper, the optimized method shows a difference of 1,941,798 cycles compared to the reference implementation. This means that applying the optimization method results in a 4% performance improvement in the key-generation process compared to the existing implementation. Skipping a single instruction in the overall process may seem trivial, but as more functions skip instructions, the performance improvement becomes more significant.

Specifically, by reducing a single instruction in a specific task process, a substantial 4% performance improvement is achieved. This is accomplished by profiling the key-generation process to identify tasks generating significant overhead and implementing optimized functions for these identified tasks. As a result, greater performance improvements can be realized.

In conclusion, when considering the entire key-generation, -signing and -verification processes of the HAWK algorithm, the overall performance improvement through T́otalŕesults shows that a performance gain of approximately 2.5% for Hawk512 and 4% for Hawk1024 is possible. These results demonstrate that performance enhancement can be achieved through profiling analysis and optimized implementation.

5. Conclusions

In this paper, an optimized implementation of the HAWK algorithm, which is currently undergoing the first round of evaluation as an electronic signature candidate, was performed. To achieve this optimized implementation, frequently used functions were analyzed through profiling analysis, and optimized functions were implemented to improve performance. This study demonstrates the importance of instruction use in optimized implementations, showing that significant performance improvements can be achieved by omitting even a single instruction. In this regard, it emphasizes the importance of profiling analysis as a critical step in the optimization process. Additionally, it was discovered that the parallel implementation of the NTT resulted in performance degradation for lower $\log n$ values, leading to optimization by applying parallel implementation only for specific $\log n$ values. As a result, this paper achieved a performance improvement of approximately 2.5% for Hawk512 and about 4% for Hawk1024. We simply optimized a few functions through profiling analysis for the key-generation process. This approach is not limited to the HAWK algorithm but can be applied to other algorithms as well. Moreover, we propose a Parallel NTT implementation method, which is expected to be applicable to other architectures where vector registers are available. Furthermore, for performance improvements in the signing and verification processes, it appears that applying an improved implementation of the SHAKE algorithm could offer further enhancements.

Author Contributions: Software, S.E.; Investigation, M.L.; Writing—original draft, S.E.; 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%).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

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

References

1. FIPS 203. Available online: https://csrc.nist.gov/pubs/fips/203/ipd (accessed on 25 July 2024).
2. FIPS 204. Available online: https://csrc.nist.gov/pubs/fips/204/ipd (accessed on 25 July 2024).
3. FIPS 205. Available online: https://csrc.nist.gov/pubs/fips/205/ipd (accessed on 25 July 2024).
4. Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schwabe, P.; Seiler, G.; Stehlé, D. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *1*, 238–268. [CrossRef]

5. Bernstein, D.J.; Hülsing, A.; Kölbl, S.; Niederhagen, R.; Rijneveld, J.; Schwabe, P. The SPHINCS+ signature framework. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 2129–2146.
6. Fouque, P.A.; Hoffstein, J.; Kirchner, P.; Lyubashevsky, V.; Pornin, T.; Prest, T.; Ricosset, T.; Seiler, G.; Whyte, W.; Zhang, Z.; et al. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. In *Submission to the NIST's Post-Quantum Cryptography Standardization Process*; 2018; Volume 36, pp. 1–75. Available online: https://falcon-sign.info/falcon.pdf (accessed on 25 July 2024).
7. Post-Quantum Cryptography: Digital Signature Schemes. Available online: https://csrc.nist.gov/projects/pqc-dig-sig/standardization/call-for-proposals (accessed on 25 July 2024).
8. 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, Sousse, Tunisia, 19–21 July 2023; Springer: Berlin/Heidelberg, Germany, 2023; pp. 417–441.
9. 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**. Available online: https://eprint.iacr.org/2022/1706 (accessed on 20 September 2024).
10. 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]
11. Ducas, L.; Postlethwaite, E.W.; Pulles, L.N.; Woerden, W.v. Hawk: Module LIP makes lattice signatures fast, compact and simple. In Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, 5–9 December 2022; Springer: Berlin/Heidelberg, Germany, 2022; pp. 65–94.
12. Haviv, I.; Regev, O. On the lattice isomorphism problem. In Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, Portland, OR, USA, 5–7 January 2014; SIAM: New Delhi, India, 2014; pp. 391–404.
13. Hoffstein, J. NTRU: A Ring Based Public Key Cryptosystem. In *Algorithmic Number Theory (ANTS III)*; Springer: Berlin/Heidelberg, Germany, 1998.
14. Pei, D.; Salomaa, A.; Ding, C. *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*; World Scientific: Singapore, 1996.
15. Seiler, G. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *Cryptol. Eprint Arch.* **2018**. Available online: https://eprint.iacr.org/2018/039 (accessed on accessed on 20 September 2024).
16. Yaman, F.; Mert, A.C.; Öztürk, E.; Savaş, E. A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1020–1025.
17. Kim, Y.; Song, J.; Seo, S.C. Accelerating Falcon on ARMv8. *IEEE Access* **2022**, *10*, 44446–44460. [CrossRef]
18. ARMv8-A Instruction Set Architecture. Available online: https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets (accessed on 25 July 2024).
19. Eum, S.; Seo, H. Improved Parallel ARIA Optimization Implementation on 64-bit ARMv8 Processor. *J. Korea Inst. Inf. Commun. Eng.* **2024**, *28*. [CrossRef]