Additive FFTs for HQC on ARM Cortex-M4, Revisited

Ming-Shing Chen[1], Tun-You Chien[1,2], Chun-Ming Chiu[1], Han-Hsuan Lin[2], Chun-Tao Peng[1] and Bo-Yin Yang[1]

[1] IIS and CITI, Academia Sinica, Taipei, Taiwan
{mschen,by}@crypto.tw,r12922085@csie.ntu.edu.tw,PengTaiwan6517@gmail.com

[2] National Tsing Hua University, Hsinchu, Taiwan
daniel0702chien@gmail.com,linhh@cs.nthu.edu.tw

Abstract. This paper presents an optimized implementation of the Hamming Quasi-Cyclic (HQC) key encapsulation mechanism, leveraging the additive fast Fourier transform (FFT) for polynomial multiplication. A primary challenge in applying FFT-based multiplication to HQC is that the polynomial degrees slightly exceed powers of two, making standard FFT approaches inefficient. To address this, we propose a new method combining the Frobenius additive FFT (FAFFT) with the Chinese Remainder Theorem (CRT) to efficiently multiply polynomials of these specific degrees. Such a combination is made possible by our new interpretation of FAFFT's `Encode` step as ring isomorphisms, from which we derive an exact formula for the modulus of any FAFFT-based polynomial multiplier.

In addition to the multiplication algorithm, we replace the Berlekamp-Massey decoder with an Extended Euclidean Algorithm (EEA) based method. The regular data flow of EEA facilitates the use of our highly optimized $\mathbb{F}_{256}$ SIMD arithmetic, leading to a faster execution speed.

Benchmarks demonstrate that our FFT-based approach significantly outperforms traditional Toom-Karatsuba methods, even at lower degrees, on the Arm Cortex-M4 platform. Our integrated optimizations result in a 19.5% and 20.4% speedups for the encapsulation and the decapsulation processes compared to the current state-of-the-art HQC-1 implementation.

Keywords: HQC · additive FFT · $\mathbb{F}_2$ polynomial multiplication · extended CRT

1 Introduction

With the growing threat of quantum computers, NIST initiated the Post-Quantum Cryptography (PQC) Standardization Process [MPR+24] in 2016. *Hamming Quasi-Cyclic (HQC)* [AAB+22, AAB+25] has been officially selected as a standard for public key encapsulation mechanism (KEM). The underlying problem of decoding quasi-cyclic codes is believed to be resistant even against quantum adversaries.

Polynomial multiplications in the ring $\mathcal{R} = \mathbb{F}_2[x]/(X^n + 1)$ are essential components in HQC. The length ranges from 18k to 58k bits, based on the chosen security level. Another important operation during the encapsulation is decoding. By optimizing these two operations, the computational cost of key generation, encapsulation, and decapsulation can be significantly reduced.

Embedded systems, such as IoT devices and smart cards, are increasingly prevalent in our daily lives. These devices often have limited computational resources, making

efficient cryptographic implementations essential. We present an optimized HQC on ARM Cortex-M4 to better enhance its usability on embedded systems.

1.1 Our Contributions

In this work, we present a new Frobenius additive fast Fourier transform (FAFFT) technique with Chinese remainder theorem(CRT) for accelerating Boolean polynomial multiplication in HQC. This method provides another perspective for Frobenius additive FFT and is particularly useful in scenarios where the degrees of polynomials are slightly greater than powers of 2.

We propose an optimized FFT-based polynomial multiplication on ARM Cortex-M4. Prior works suggested that the FFT-based method is inferior in HQC-1. We have shown that the FFT-based method is indeed the better choice for all security levels on ARM Cortex-M4.

Besides polynomial multiplications, we also present an efficient implementation for the Reed-Solomon (RS) coding algorithm in HQC. The implementation is based on a constant-time extended Euclidean algorithm (EEA).

Compared with the previous state-of-the-art implementation, our HQC implementation achieves 19.5% and 20.4% speedups in encapsulation and decapsulation on HQC-1, 15.9% and 15.9% on HQC-3.

Source code. The source code for all our implementations is available under CC0 copyright-waiver at `https://github.com/ChunTaoPengim/HQC_with_additive_fft_M4`.

1.2 Related Works

Boolean polynomial multiplication. Gao and Mateer presented their additive FFT (addFFT) in 2010 [GM10]. Van der Hoeven and Larrieu [vdHL17] proposed using the Frobenius mapping to further accelerate the algorithm. These addFFTs were used for Boolean polynomial multiplications in [CCK+17], [CCK+18], and [LCK+18], showing that FFT-based methods are superior when the polynomial degrees are large enough.

Recently, Chen, Chiu, Peng, and Yang [CCPY26] presented an additive FFT+CRT technique for processing polynomials of length $2^i + c$ in the context of HQC. While they introduced a CRT-based decomposition, their reliance on standard additive FFTs restricts the algebraic flexibility of the underlying quotient rings, meaning their results do not straightforwardly apply to the FAFFT.

Algorithmically, we fundamentally shift this paradigm by formalizing FAFFT's Encode step as a strict ring isomorphism. This provides an exact, explicit formula for the modulus $Q(x)$ of any FAFFT-based multiplier. By actively searching for a $Q(x)$ with a minimal Hamming weight, we significantly reduce the computational cost of the reverse CRT map.

Regarding implementation, [CCPY26] targets platforms equipped with powerful single-instruction-multiple-data (SIMD) instruction sets. In contrast, we focus on the instruction-limited Arm Cortex-M4 architecture. This difference in available instructions leads to entirely different optimization strategies and design choices.

HQC on embedded systems. Polynomial multiplications (polymul) performance for schemes like BIKE and HQC on embedded systems is a highly competitive and active area of research. Early aFFT-based implementations [CCK21] demonstrated superiority to Karatsuba for relatively short polymuls on instruction-limited platforms. This performance lead was subsequently challenged in [CGKT22], which leveraged the standard 32×32→64-bit *integer* multiplication instructions for carry-less polynomial multiplication. Their optimized Karatsuba implementation outperformed the previous FFT methods

for polynomial degrees below 20k bits, establishing a clear performance crossover point. Later, [LJKH25, JLK+25] focused on FFT-based multiplication again and further accelerated FFT implementations that outperformed previous implementations in the HQC scheme within the embedded context.

1.3 Organization of This Paper

We introduce HQC, targeting architectures, and base polynomial multiplications in Section 2. Section 3 reviews additive FFT, Frobenius additive FFT multiplication for Boolean polynomials. Section 4 presents new FAFFT-based multiplication techniques tailored for the polynomial lengths in HQC. Section 5 presents the RS codec with constant-time extended GCD techniques. Section 6 shows our optimized implementation of previous techniques on the ARM Cortex-M4 platform. Section 7 presents performance evaluation.

2 Preliminaries

2.1 HQC

HQC (Hamming Quasi-Cyclic) [AAB+25] is a code-based post-quantum key encapsulation mechanism (KEM), which was selected to be standardized by NIST recently. Its core component is a public-key encryption scheme HQC-PKE. The KEM HQC-KEM is derived from HQC-PKE using the Fujisaki-Okamoto transformation [FO99] following the HHK framework [HHK17]. The encapsulation key $\mathsf{ek}_{\mathsf{KEM}}$ is the same as the encryption key $\mathsf{ek}_{\mathsf{PKE}}$. Due to the re-encryption operation in FO transform, the decapsulation key $\mathsf{dk}_{\mathsf{KEM}}$ comprises $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}}, \sigma, \mathsf{seed}_{\mathsf{KEM}})$ where $\mathsf{dk}_{\mathsf{PKE}}$ is the decryption key, σ is randomness for the shared key of KEM, and $\mathsf{seed}_{\mathsf{KEM}}$ is the 32-byte seed for all elements in HQC-KEM.

Algorithm 1 shows the three main components in HQC-PKE. Its main computation is multiplication in the ring

$$\mathcal{R} := \mathbb{F}_2[x]/(x^n - 1) \ . \tag{1}$$

The parameter n and other specifications for 3 different security levels are listed in Table 1. When $\mathbb{F}_2$ polynomials in the ring are sparse, extra parameters (w, w_r, w_e) specify their Hamming weight of coefficients. HQC-PKE specifies an error correcting code $\mathcal{C}$ (see Subsubsection 2.1.1 for details). A parameter ℓ denotes the truncated bits when transforming polynomials in $\mathcal{R}$ into $\mathbb{F}_2$ vectors used in its code $\mathcal{C}$.

Throughout this paper, we always process the multiplication in the ring $\mathcal{R} := \mathbb{F}_2[x]/(x^n - 1)$ as follows: 1) Lift the ring elements in $\mathcal{R}$ into polynomials in $\mathbb{F}_2[x]$ and process polynomial multiplication in $\mathbb{F}_2[x]$. 2) Reduce the products in $\mathbb{F}_2[x]$ by adding the terms of degree $\geq n$ into the terms of lower degrees and map the results back to elements in $\mathcal{R}$. Hence, we focus on multiplying polynomials in $\mathbb{F}_2[x]$ in the following contents.

Table 1: Parameter of HQC-KEM.

	n	w	$w_r = w_e$	$\lvert\mathsf{ek}_{\mathsf{KEM}}\rvert$	$\lvert\mathsf{dk}_{\mathsf{KEM}}\rvert^{\dagger}$	$\lvert\mathsf{c}_{\mathsf{KEM}}\rvert$	$\lvert K\rvert$
HQC-1	17 669	66	75	2241	2321	4433	32
HQC-3	35 851	100	114	4514	6402	8978	32
HQC-5	57 637	131	149	7237	7333	14 421	32

Sizes of keypair $(\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{KEM}})$, ciphertext $\mathbf{c}_{\mathsf{KEM}}$ and shared key K are reported in bytes.

† Here shows the size of the default mode. $\lvert\mathsf{dk}_{\mathsf{KEM}}\rvert$ are all 32 bytes in compressed mode.

Algorithm 1 Main components in HQC-PKE.

Global parameters: $(n, w, w_r, w_e, \ell, \mathcal{C})$

HQC-PKE.Keygen($\mathsf{seed}_{\mathsf{PKE}}$):
- $(\mathsf{dk}_{\mathsf{PKE}}, \mathsf{seed}_{\mathsf{PKE.ek}}) \leftarrow \mathsf{HASH.I}(\mathsf{seed}_{\mathsf{PKE}})$
- $(y, x) \xleftarrow{\mathsf{dk}_{\mathsf{PKE}}} \mathcal{R}_w \times \mathcal{R}_w$
- $h \xleftarrow{\mathsf{seed}_{\mathsf{PKE.ek}}} \mathcal{R}$
- $\mathsf{ek}_{\mathsf{PKE}} = (\mathsf{seed}_{\mathsf{PKE.ek}}, x + h \cdot y)$
- **return** $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}})$

HQC-PKE.Encrypt($\mathsf{ek}_{\mathsf{PKE}}, m, \theta$):
- $(r_2, e, r_1) \xleftarrow{\theta} (\mathcal{R}_{w_r}, \mathcal{R}_{w_e}, \mathcal{R}_{w_r})$
- $h \xleftarrow{\mathsf{ek}_{\mathsf{PKE}}} \mathcal{R}$
- $u \leftarrow r_1 + h \cdot r_2$
- $v \leftarrow \mathcal{C}.\mathsf{Encode}(m) + \mathrm{tr}(s \cdot r_2 + e, \ell)$
- **return** $\mathbf{c}_{\mathsf{PKE}} = (u, v)$

HQC-PKE.Decrypt($\mathsf{dk}_{\mathsf{PKE}}, \mathbf{c}_{\mathsf{PKE}} = (u, v)$):
- $y \xleftarrow{\mathsf{dk}_{\mathsf{PKE}}} \mathcal{R}_w$
- **return** $\mathcal{C}.\mathsf{Decode}(v - \mathrm{truncate}(u \cdot y, \ell))$

2.1.1 Concatenated Reed-Muller and Reed-Solomon Codes

HQC adopts a concatenated coding scheme that combines a duplicated Reed-Muller (RM) code as the inner code and a Reed-Solomon (RS) code as the outer code. (See [LC04] for the details of the RS and RM code.) The code parameters are specified in [AAB+25], and the method is described in [AGZ].

Reed-Solomon code. The Reed-Solomon code defined in HQC as RS$[n_1, k]$ code over $\mathbb{F}_{256} := \mathbb{F}_2[x]/(x^8+x^4+x^3+x^2+1)$, i.e., the `0x11D` $\mathbb{F}_{256}$. Here, n_1 is the block length. The number of information symbols is k, and the number of parity-check symbols is $n_1 - k = 2\delta$, where δ is the error correcting capability. The specific parameters (n_1, k, δ) are $(46, 16, 15)$, $(56, 24, 16)$, and $(90, 32, 29)$ for HQC-1, HQC-3, and HQC-5, respectively.

HQC uses a systematic RS code, where the original encoded message is a part of its codeword. A codeword is treated as a multiple of a predefined generator polynomial, i.e., the BCH view. Let α be a primitive element in $\mathbb{F}_{256}$. The generator polynomial is given by $g(x) = (x + \alpha)(x + \alpha^2) \cdots (x + \alpha^{2\delta})$. As described in the spec [AAB+25], the RS decoder uses syndrome decoding. The syndromes are evaluated from the (corrupted) codeword polynomial on the roots of the generator polynomial. The decoder fixes the corrupted codeword polynomial based on the syndromes.

Reed-Muller code. In HQC, the RM encoder first encodes every byte of the previous RS codeword into a 128-bit vector, i.e., a $(128, 8, 64)$ RM code over $\mathbb{F}_2$. It then duplicates the previous vector 3 or 5 times to form the final codeword. In the decryption process, the coder first sums the (corrupted) duplicated codewords and applies the maximum likelihood decoding algorithm to recover the codeword.

2.1.2 Side-Channel Resistance

We keep our implementations with a time constancy property, in which memory access and control flow are independent of secret values, to prevent timing side-channel leakage. We do not specialize our polynomial multiplier for the sparse polynomials in HQC because the degrees of the terms of the sparse polynomials will leak.

2.2 Targeting Architectures

We evaluate our proposed approaches on the ARM Cortex-M4 processor. The M4 has already been used as a representative for microcontrollers in numerous publications about efficient implementations of post-quantum cryptography and has also been selected as the reference platform for embedded benchmarks by the NIST.

The ARM Cortex-M4 is a 32-bit RISC processor implementing the `ARMv7E-M` ISA [Arm20].

It provides the programmer with thirteen 32-bit general-purpose registers, and a further link register can be used for computations when its content is preserved on the stack. The optional floating-Point (FP) unit comes with 32 additional 32-bit registers. Single 32-bit memory access usually takes 2 cycles. When using the `STM` instruction for loading consecutive n 32-bit words, it can be as fast as n+1 cycles. Storing n consecutive 32-bit words take n+1 cycles as well as loading. With a latency of one cycle to move data between a general-purpose register and a floating-point register, we cache general-purpose registers in the FP registers in our work.

In this work, we use the multiply-and-accumulate instruction (`umlal`) exceedingly. The instruction `umlal rdLo, rdHi, rn, rm` multiplies two 32-bit integers in rn and rm and then adds the 64-bit product to the values of two 32-bit registers (`rdHi,rdLo`) in one clock cycle. The `umull` instruction provides the same 64-bit product without accumulation. It's worth noting that many instructions on the ISA allow for the shift of one of the operands without latency overhead.

2.3 Base Polynomial Multiplication in Radix-16 Representation

Polynomial multiplication via radix-16. An efficient method for polynomial multiplication in $\mathbb{F}_2[x]$ leverages integer arithmetic using the radix-16 representation [CC21]. In this work, we denote the radix-16 representation of a polynomial $P(x)$ as $\tilde{P}$. A degree-7 polynomial $A_0(x) = \sum_{i=0}^{7} a_i x^i$ is mapped to a 32-bit integer $\tilde{A_0} = \sum_{i=0}^{7} a_i 2^{4i}$. Multiplying two such integers $\tilde{A_0} \cdot \tilde{B_0}$ results in a product where the coefficient c_i of $C_0(x) = A_0(x) \cdot B_0(x)$ resides at the $(4i)$-th bit index of $\tilde{C_0}$. The interference from cross-terms is avoided due to the spacing redundancy of the radix-16 representation, and the result is recovered by applying a mask (a logical `AND` operation). The process of 8-bit $\mathbb{F}_2[x]$ polynomial multiplication with radix-16 representation is shown in Figure 1.

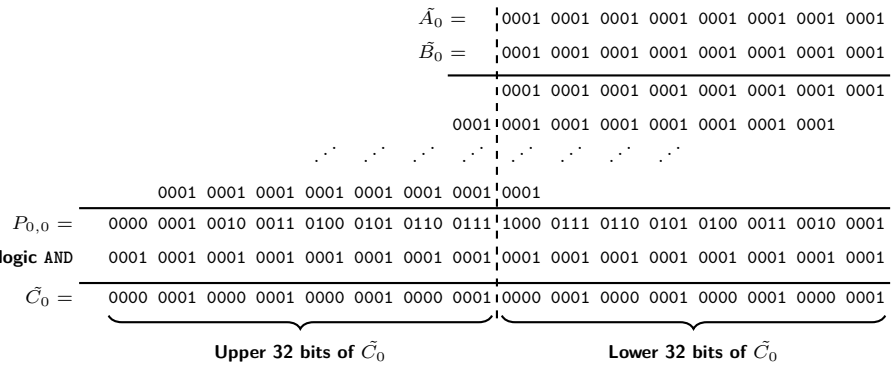

Figure 1: 8-bit $\mathbb{F}_2[x]$ polynomial multiplication with radix-16 representation.

Extension to 32-bit polynomials. For multiplying 32-bit polynomials, [CGKT22] proposed splitting inputs $A(x), B(x)$ into four 8-bit partial polynomials A_i, B_j (for $i, j \in \{0, \ldots, 3\}$). The multiplication $C(x) = A(x) \cdot B(x)$ is computed by summing partial products $\tilde{A_i} \cdot \tilde{B_j}$ at their respective offsets, followed by the extraction of the valid bits. Advanced implementations further optimize this using recursive Karatsuba or Toom-Cook decomposition [JLK+25].

Data format conversion and packing. To reduce register pressure, four 8-bit polynomials are packed into a single 32-bit register A_{pack}. Converting the data format between a standard bit sequence and this packed radix-16 representation involves bit permutations. [JLK+25] achieved this by employing three `SWAPMOVE` operations [Bih97], requiring a total of 12 instructions (averaging 3 instructions per 8-bit polynomial). Once packed, each of

the four radix-16 representations, $\tilde{A}_i$, can be computed via a single masking operation with `0x11111111`:

$$\tilde{A}_i = (A_{\text{pack}} \gg k_i) \texttt{ AND } 0\text{x}11111111, \quad \text{where } (k_0, k_1, k_2, k_3) = (0, 2, 1, 3).$$

2.4 Algebraic Background for Polynomial Rings

To establish the algebraic background for our optimizations, we briefly review standard concepts in abstract algebra, referring readers to classical texts such as Dummit and Foote [DF03] for comprehensive proofs.

Homomorphisms and the First Isomorphism Theorem. Let R and S be rings. A ring homomorphism $\eta : R \to S$ is a map preserving the ring operations. The kernel of η, denoted $\ker(\eta) = \{r \in R \mid \eta(r) = 0\}$, forms an ideal in R. The First Isomorphism Theorem states that the quotient ring $R/\ker(\eta)$ is strictly isomorphic to the image of η, i.e., $R/\ker(\eta) \cong \text{Im}(\eta)$.

Ideals in $\mathbb{F}_2[x]$. The polynomial ring over a field, such as $\mathbb{F}_2[x]$, is a Principal Ideal Domain (PID). This implies that every ideal $I \subseteq \mathbb{F}_2[x]$ is generated by a single polynomial, denoted as $I = \langle Q(x) \rangle$. Specifically, the kernel of any homomorphism from $\mathbb{F}_2[x]$ is generated by a unique minimal-degree polynomial $Q(x)$.

The Chinese Remainder Theorem (CRT). For pairwise coprime polynomials $P_1(x), P_2(x) \in \mathbb{F}_2[x]$, the ideals generated by them are comaximal. The CRT states that there is a ring isomorphism: $\mathbb{F}_2[x]/\langle P_1(x)P_2(x) \rangle \cong \mathbb{F}_2[x]/\langle P_1(x) \rangle \times \mathbb{F}_2[x]/\langle P_2(x) \rangle$. This theorem allows us to compute operations in a larger ring by decomposing them into independent computations in smaller quotient rings.

3 Polynomial Multiplication in $\mathbb{F}_2[x]$

This section first reviews existing polynomial multiplication algorithms over $\mathbb{F}_2[x]$. We then introduce the additive fast Fourier transform (additive FFT) as the fundamental component for FFT-based multiplication. Finally, we present the Frobenius additive FFT (FAFFT) [CCK+18], leveraging it to efficiently multiply long polynomials in $\mathbb{F}_2[x]$.

Polynomial multiplication algorithms over $\mathbb{F}_2[x]$ are typically selected based on a trade-off between polynomial degree and hardware capabilities. While base multiplications leverage word-sized hardware instructions, divide-and-conquer methods like Karatsuba and Toom-Cook are preferred for medium-sized operands. For larger degrees, FFT-based algorithms offer the lowest asymptotic complexity. The practical crossover points between these categories are not fixed but are instead a function of the underlying architecture's instruction set.

3.1 Karatsuba and Toom Cook

While the official HQC implementation employs a hybrid Toom-Karatsuba approach [Knu97], this method exhibits higher asymptotic complexity than FFT-based algorithms.

The Karatsuba algorithm reduces the multiplication of $(A_0 + A_1x^n) \cdot (B_0 + B_1x^n)$ to three smaller products: $A_0 \cdot B_0$, $A_1 \cdot B_1$, and $(A_0 + A_1) \cdot (B_0 + B_1)$ by $(A_0 + A_1x^n) \cdot (B_0 + B_1x^n) = A_0B_0 + [(A_0 + A_1) \cdot (B_0 + B1) + A_0B_0 + A_1B_1]x^n + A_1B_1x^{2n}$. This process is applied recursively until reaching the base multiplication case, achieving an asymptotic complexity of $O(n^{\log_2^3})$.

The Toom-Cook algorithm follows an evaluation-interpolation framework: input polynomials are evaluated at specific points, multiplied point-wise, and finally interpolated through a pre-defined matrix to obtain the result.

3.2 Additive Fast Fourier Transform

This section introduces the additive FFT [Can89, GM10, LCH14, LANHC16], which efficiently evaluates a polynomial over some additive subgroups of the coefficient field.

Binary field and Cantor basis.

We first define a series of binary fields, which are used in the additive FFT algorithm in this paper. The binary extension fields of $\mathbb{F}_{2^m}$, where m is a power of 2, are extended from $\mathbb{F}_2$ as:

$$\begin{aligned} \mathbb{F}_{2^2} &\coloneqq \mathbb{F}_2[x_0]/(x_0^2+x_0+1) & \mathbb{F}_{2^{16}} &\coloneqq \mathbb{F}_{2^8}[x_3]/(x_3^2+x_3+x_2x_1x_0) \\ \mathbb{F}_{2^4} &\coloneqq \mathbb{F}_{2^2}[x_1]/(x_1^2+x_1+x_0) & \mathbb{F}_{2^{32}} &\coloneqq \mathbb{F}_{2^{16}}[x_4]/(x_4^2+x_4+x_3x_2x_1x_0) \\ \mathbb{F}_{2^8} &\coloneqq \mathbb{F}_{2^4}[x_2]/(x_2^2+x_2+x_1x_0) \end{aligned} \tag{2}$$

An element in $\mathbb{F}_{2^{32}}$ is represented as a degree-1 multivariate polynomial in $\mathbb{F}_{2^{16}}[x_4]$ or an unsigned 32-bit integer.

The Cantor basis $\{v_0, \ldots, v_{31}\}$ of $\mathbb{F}_{2^{32}}$ is a set of elements that satisfies the recurrence relation

$$v_i = v_{i+1}^2 - v_{i+1} \quad \text{with } v_0 = 1 \ . \tag{3}$$

In this work, we specify v_{31} as $v_{31} = 1+x_0x_1+x_0x_1x_2+x_3+x_0x_1x_3+x_0x_2x_3+x_1x_2x_3+x_4+x_0x_1x_4+x_2x_4+x_0x_2x_4+x_1x_2x_4+x_0x_1x_2x_4+x_0x_3x_4+x_2x_3x_4+x_0x_1x_2x_3x_4$. Beyond the multivariate polynomial form, we represent elements in $\mathbb{F}_{2^{32}}$ using an integer index. For any integer $i \in [0, 2^{32}-1]$ with binary expansion $i = \sum_{j=0}^{31} i_j 2^j$, the corresponding field element is defined as:

$$\underline{i} = \sum_{j=0}^{31} i_j \cdot v_j \in \mathbb{F}_{2^{32}}. \tag{4}$$

Under this representation, the field is succinctly denoted as $\mathbb{F}_{2^{32}} = \{\underline{0}, \underline{1}, \ldots, \underline{2^{32}-1}\}$, where $\underline{0} = 0$ and $\underline{1} = 1$.

Vanishing polynomials and polynomial basis. A sequence of subspaces is constructed from the Cantor basis as $V_0 = \{0\}$ and

$$V_i = \mathsf{span}\{v_0, \ldots, v_{i-1}\} = \{0, \ldots, \sum_{j=0}^{i-1} v_j\} = \{0, \ldots, \underline{2^i-1}\} \quad \text{for } i \in \{1, \ldots, 32\} \ .$$

For each subspace V_i, the vanishing polynomial $s_i(x) \in \mathbb{F}_2[x]$ of degree 2^i is defined such that

$$s_i(x) = \Pi_{j \in V_i}(x-j) = x(x-1)(x-\underline{2})\cdots(x-\underline{2^i-1}) \ , \tag{5}$$

mapping all elements in V_i to zero.

These polynomials exhibit several properties essential for the additive FFT [GM10, LCH14]:

$$\text{Linearity: } s_i(a+b) = s_i(a) + s_i(b) \ . \tag{6}$$

$$\text{Recursion: } s_i(x) = s_{i-1}(s_1(x)) \text{ for } i > 1 \ . \tag{7}$$

$$\text{Evaluation of } s_i: \ s_i(\underline{j}) = \underline{j \gg i} \ . \tag{8}$$

Specifically, the generating relation (3) simplifies to the evaluation of $s_1(x)$ at a basis element: $s_1(v_i) = v_i^2 - v_i = v_{i-1}$ for $i \in \{1, \ldots, 31\}$.

Lin, Chung, and Han [LCH14] proposed an additive FFT to evaluate polynomials over the basis $\{1, X_1, X_2, X_3, \ldots\}$, where X_i of degree i is defined from vanishing polynomials s_i as

$$X_i = \prod_{j \geq 0} s_j^{i_j} \quad \text{, where } i = \sum_{j \geq 0} i_j \cdot 2^j \text{ and } i_j \in \{0, 1\} \ . \tag{9}$$

This gives us an alternative way of representing polynomials, instead of linear combinations of monomials $1, x, x^2, \ldots$. In other words, any $f \in \mathbb{F}_{2^m}[x]_{<n}$ can be stored as the coefficients $\overline{f}_0, \ldots, \overline{f}_{n-1}$ such that $f = \sum_{j=0}^{n-1} \overline{f}_j X_j$. The main advantage of this representation is that any Euclidean division with the vanishing polynomials s_i is basically trivialized. Algorithm 2 details the procedure for converting a polynomial f from the monomial basis to this polynomial basis.

Algorithm 2 Basis conversion (`BasisCvt`) algorithm.

```
1: BasisCvt(f(x) ):                          ▷ f = f_0 + ... + f_{n-1} x_{n-1} in monomial basis.
2:     if deg(f) ≤ 1 then
3:         return f_0 + f_1 X_1.
4:     Find the largest i = 2^j s.t. i is a power of 2 and deg(s_i) < deg(f).
5:     Let y = s_i(x).
6:     Compute ĥ(y) = h(x, y) = h_0(x) + ... + h_t(x) y^t s.t. deg(h_k) < deg(s_i) and h(x, y) = f(x).
7:     Compute h'(Y) = q_0(x) + q_1(x) X_{2^i} + ... + q_t(x) X_{2^i t} ← BasisCvt(ĥ(y)).
8:     for all coefficients q_j(x) in h'(Y) do
9:         Compute g_{j·2^i} + g_{j·2^i+1} X + ... + g_{(j+1)·2^i−1} X_{2^i−1} ← BasisCvt(q_j(x)).
10:    return g(X) = g_0 + g_1 X_1 + ... + g_{n-1} X_{n-1}.
```

Additive FFT for polynomial evaluation. The additive FFT (`addFFT`) evaluates a polynomial $F \in \mathbb{F}_{2^m}[x]$ (given in its novel representation $\overline{F}$) over an affine subspace $a + V_i$, returning a sequence of evaluations

$$\texttt{addFFT}(\overline{F}, a + V_i) = \big(F(u) \mid u \in a + V_i\big) = \big(F(a+0), \ldots, F(a + \underline{2^i - 1})\big).$$

Algorithm 3 Additive Fast Fourier Transform (`addFFT`) algorithm.

```
1: addFFT(F̄, a + V_i ):                      ▷ F is a polynomial of degree n − 1 = 2^i − 1.
2:     if i = 0 then
3:         return (F̄_0)
4:     Let L̄_j = F̄_j, H̄_j = F̄_{j+n/2} for 0 ≤ j < n/2
                                   ▷ L and H are polynomials of degrees n/2 − 1 s.t. F = L + s_{i-1} H.
5:     Compute L̄' ← L̄ + s_{i-1}(a) H̄
6:     Compute H̄' ← L̄ + (s_{i-1}(a) + 1) H̄ as L̄' + H̄
7:     return (addFFT(L̄', a + V_{i-1}), addFFT(H̄', a + v_{i-1} + V_{i-1}))
```

For a polynomial $F = L + s_{i-1}H$ of degree $n = 2^i - 1$, one can show that $\overline{L}_j = \overline{F}_j, \overline{H}_j = \overline{F}_{j+n/2}$ for $0 \leq j < n/2$, and that $\overline{L + bH} = \overline{L} + b\overline{H}$ for all constant b. The `addFFT` works by recursively reducing the evaluation over $a + V_i$ into two sub-problems of size $n/2$:

$$\texttt{addFFT}(\overline{L} + s_{i-1}(a)\overline{H}, a + V_{i-1}) \text{ and } \texttt{addFFT}(\overline{L} + (s_{i-1}(a) + 1)\overline{H}, a + v_{i-1} + V_{i-1}) \ .$$

This decomposition is implemented via the `Butterfly` operation (Figure 2), which requires only a single multiplication by $s_{i-1}(a)$ per unit. Across $\log_2 n$ stages, the algorithm yields the evaluation vector $(f(u) \mid u \in a + V_i)$. We outline the whole procedure in Algorithm 3.

The `addFFT` requires $n/2$ operations in each of its $\log_2 n$ stages, leading to the overall complexity of $O(n \log n)$. However, it requires the input to be already in the novel representation. If only the monomial representation is available, the total cost would be dominated by the conversion step `BasisCvt`, which takes $O(n \log n \log \log n)$ operations [LANHC16].

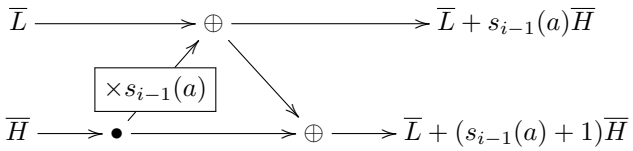

Figure 2: The `Butterfly` operation in the additive FFT.

3.3 Frobenius Additive FFT for Polynomial Multiplication in $\mathbb{F}_2[x]$

Consider the homomorphism $\mathbb{F}_2[x] \to \mathbb{F}_{2^m}[x]/\langle s_l(x) + v_{m/2}\rangle$ that sends a polynomial F to its remainder modulo $s_l(x) + v_{m/2}$ where $l = \log_2(2n/m)$. In [CCK+18], it was (implicitly but effectively) shown that, when the domain is restricted to $\mathbb{F}_2[x]_{<2n}$, the map is injective (and therefore bijective, because $\dim_{\mathbb{F}_2}(\mathbb{F}_2[x]_{<2n}) = \dim_{\mathbb{F}_2}(\mathbb{F}_{2^m}[x]/\langle s_l(x) + v_{m/2}\rangle) = 2n$). In other words, each remainder $F' \in \mathbb{F}_{2^m}[x]/\langle s_l(x) + v_{m/2}\rangle$ has a unique preimage of length-$2n$. Thus, as long as the product lies inside $\mathbb{F}_2[x]_{<2n}$ (e.g., if the two multiplicands are inside $\mathbb{F}_2[x]_{<n}$), it's possible to compute multiplication in $\mathbb{F}_2[x]$ efficiently by reducing it to multiplication in $\mathbb{F}_{2^m}[x]/\langle s_l(x) + v_{m/2}\rangle$, which is naturally handled by addFFT. The only restriction is that $l < m/2$, which is necessary for $v_{m/2+l}$ to lie in $\mathbb{F}_{2^m}$ so that $s_l(x) + v_{m/2} = s_l(x) + s_l(v_{m/2+l})$ splits.

This is known as Frobenius Additive FFT (FAFFT), and the invertible map $F \mapsto F' = F \bmod (s_l(x)+v_{m/2})$ is known as the `Encode` step. In Subsubsection 4.2.1, we offer another view of the theory behind this technique.

As pointed out in [CCK+18], the `Encode` step is exactly the same as the first $\log_2 m$ stages of a truncated $\texttt{addFFT}(A, \tilde{\Sigma})$, where $\Sigma \subset \tilde{\Sigma} = v_{l+m/2} + V_{l+\log_2 m}$. From multiplying constants of the first $\log_2 m$ stages in Algorithm 3, the `Encode` (under the novel representation domain) computes

$$\overline{F'}_i = \sum_{j=0}^{m-1} X_j(v_{m/2})\overline{F}_{i+j(2n/m)} \ . \tag{10}$$

Remarkably, each $\overline{F'}_i$ only depends on m coefficients (namely $\overline{F}_{i+j(2n/m)}$ for $0 \le j < m$), and the sequence of coefficients $X_j(v_{m/2})$ is fixed. Consequently, the linear map (and its inverse) can be implemented as $2n/m$ parallel instances of multiplication with a constant m-by-m matrix, instead of one multiplication by a $2n$-by-$2n$ matrix.

Now, we are ready to describe the algorithm of FAFFT-based polynomial multiplication in its entirety.

FAFFT for polynomial multiplication. Given two polynomials $A, B \in \mathbb{F}_2[x]_{<n}$, the FAFFT method computes their product $C = A \cdot B \in \mathbb{F}_2[x]_{<2n}$ as follows:

1. FAFFT:
   (a) Transform to the novel representation (after zero padding): $\overline{A}, \overline{B} \in \mathbb{F}_2^{2n} \leftarrow \texttt{BasisCvt}(A), \texttt{BasisCvt}(B)$.
   (b) Compute $A', B' \in \mathbb{F}_{2^m}[x]/\langle s_l(x)+v_{m/2}\rangle$: $\overline{A'}, \overline{B'} \in \mathbb{F}_{2^m}^{2n/m} \leftarrow \texttt{Encode}(\overline{A}), \texttt{Encode}(\overline{B})$.
   (c) Evaluation on Σ: $\hat{A}, \hat{B} \in \mathbb{F}_{2^m}^{2n/m} \leftarrow \texttt{addFFT}(\overline{A'}, \Sigma), \texttt{addFFT}(\overline{B'}, \Sigma)$.
2. Pointwise multiplication: $\hat{C} \in \mathbb{F}_{2^m}^{2n/m} \leftarrow \hat{A} \otimes \hat{B}$.
3. Inverse FAFFT:
   (a) Recover $C' \in \mathbb{F}_{2^m}[x]/\langle s_l(x) + v_{m/2}\rangle$: $\overline{C'} \in \mathbb{F}_{2^m}^{2n/m} \leftarrow \texttt{addFFT}^{-1}(\hat{C}, \Sigma)$.

(b) Recover $C \in \mathbb{F}_2[x]_{<2n}$ as the unique preimage of C': $\overline{C} \in \mathbb{F}_2^{2n} \leftarrow \texttt{Encode}^{-1}(\overline{C'})$.

(c) Transform back to monomial representation: $C \in \mathbb{F}_2[x]_{<2n} \leftarrow \texttt{BasisCvt}^{-1}(\overline{C})$.

4 Polynomial Multiplication in HQC

In this section, we introduce a new method for Boolean polynomial multiplications in HQC. In HQC-1 and HQC-3, the polynomial lengths are $n = 17669$ and 35851, both slightly larger than powers of 2, making the direct usage of the FFT-based method inefficient. To handle this problem, we present multiplications via FAFFT with CRT.

4.1 Review of Polynomial Multiplications in HQC

Previous works have shown numerous methods for dealing with the parameters. Without loss of generality, we focus on the HQC-1 parameter set where $n = 17669$:

Toom-Karatsuba approach. From the official package [AAB+25], the authors have proposed doing multiplications with Toom-Cook and Karatsuba in the reference implementations. They also provided AVX2 optimized implementations based on the method with dedicated carry-less multiplication instructions. On embedded systems, [JLK+25] showed that with the radix-16 technique [CC21, CGKT22], this approach also yields strong performance on platforms with no such instructions.

Naive FAFFT. Another approach is to zero-pad the polynomials to make the lengths suitable for the FFT-based approach. In [RLC+25] and [LJKH25], the authors have extended the polynomial lengths to be powers of 2. In the HQC-1 scenario, this means considering degrees of polynomials to be $n = 32768$.

FAFFT with cross-term. However, zero-padding is inefficient due to the computations on additional degrees. In [CCPY26], the polynomials are separated into two parts $f(x) = f_l(x) + x^{16384} f_h(x)$, $g(x) = g_l(x) + x^{16384} g_h(x)$. The authors only conducted FAFFT on multiplications of the lower degrees and performed Karatsuba for the rest. Another method to handle additional $f_h(x)$ was proposed in [JLK+25], called hybrid Karatsuba FAFFT. They used FAFFT to not only compute $f_l g_l$ but also $(f_l + f_h)(g_l + g_h)$. After computing $f_h \cdot g_h$, the cross-terms $f_h \cdot g_l + f_l \cdot g_h = (f_l + f_h)(g_l + g_h) - f_l \cdot g_l - f_h \cdot g_h$ were recovered exactly as in Karatsuba.

Additive FFT with CRT. There have already been ways that combine FFT-based multiplications with CRT to solve the problem. In [CCPY26], they design a method to compute polynomials with degrees $n = 2^i + c$, where 2^i is the largest power of 2 less than the polynomial degree, and c is the remaining part. The core idea is to decompose the larger problem into smaller, more manageable sub-problems using CRT:

$$\mathbb{F}_{2^{16}}[x]/\langle s_k x^{c/8} \rangle \xleftrightarrow{\text{CRT}} \mathbb{F}_{2^{16}}[x]/\langle s_k \rangle \times \mathbb{F}_{2^{16}}[x]/\langle x^{c/8} \rangle \ .$$

Multiplications in $\mathbb{F}_{2^{16}}[x]/\langle s_k \rangle$ was performed by additive FFT, and $\mathbb{F}_{2^{16}}[x]/\langle x^{c/8} \rangle$ used Karatsuba for the multiplications.

4.2 Combination of FAFFT and Chinese Remainder Theorem

In this section, we leverage the CRT to decompose the multiplication in HQC to avoid the zero-padding overhead in the FAFFT-based approach. Specifically, we extend the Additive FFT with CRT method in [CCPY26] to the FAFFT. Since applying CRT requires explicit knowledge of the underlying FAFFT modulus $Q(x)$, we utilize the algebraic foundations established in Subsection 2.4 to formally characterize the FAFFT's `Encode` step as a ring

homomorphism. This formalization directly provides the explicit formula for the modulus $Q(x)$, making the required CRT mapping straightforward.

Moreover, we show that by selecting the parameter in the FAFFT setup, we can construct a $Q(x)$ with a minimal Hamming weight. This sparsity significantly reduces the cost of the reverse CRT map without introducing any overhead to the underlying FAFFT operations.

4.2.1 FAFFT's `Encode` as Ring Isomorphism

To derive the explicit formula for the modulus $Q(x)$, we first formalize the `Encode` step. Recall from Subsection 3.3 that the `Encode` step essentially computes the homomorphism $\eta : \mathbb{F}_2[x] \to \mathbb{F}_{2^m}[x]/\langle s_l(x)+v_{m/2}\rangle$ by mapping a polynomial F to $\eta(F) = F \bmod (s_l+v_{m/2})$. As reviewed in Subsection 2.4, its kernel $\ker(\eta)$ is an ideal in $\mathbb{F}_2[x]$. Since $\mathbb{F}_2[x]$ is a PID, the fundamental observation is that $\ker(\eta)$ is generated by a specific polynomial $Q(x) \in \mathbb{F}_2[x]$. By the First Isomorphism Theorem, this directly establishes the exact ring isomorphism $\mathbb{F}_2[x]/\langle Q(x)\rangle \cong \mathbb{F}_{2^m}[x]/\langle s_l(x) + v_{m/2}\rangle$. Furthermore, since the map is bijective when restricted to $\mathbb{F}_2[x]_{<2n}$, it implies that $\deg Q(x) = 2n$.

For basic CRT applications, this algebraic characterization already allows us to determine $Q(x)$ empirically. By computing the unique preimage $P \in \mathbb{F}_2[x]_{<2n}$ of the evaluated element $\eta(x^{2n})$, we can immediately conclude that $Q(x) = x^{2n} - P$, because $\eta(x^{2n} - P) = \eta(x^{2n}) - \eta(P) = 0$. However, an explicit algebraic formula is desirable to actively minimize the Hamming weight of the modulus Q. In the following, we determine Q by introducing a parameter b into the isomorphism construction, which grants us the freedom to optimize the reverse CRT map.

Recall that `addFFT` gives us an efficient way to compute multiplication in the quotient ring of the form $\mathbb{F}_{2^m}[x]/\langle s_l(x) - s_l(a)\rangle$ for any $a \in \mathbb{F}_{2^m} = V_m$, or equivalently, $\mathbb{F}_{2^m}[x]/\langle s_l(x) - b\rangle$ for $b = s_l(a) \in V_{m-l}$. Let $\mu_b(y) \in \mathbb{F}_2[y]$ denote the minimal polynomial having b as a root, which formally generates the kernel of the evaluation map $y \mapsto b$, i.e., $\ker(y \mapsto b) = \langle \mu_b(y)\rangle$. Now, if $\mu_b(y)$ is of degree m, then $\mathbb{F}_2[y]/\langle \mu_b(y)\rangle \cong \mathbb{F}_{2^m}$ under this evaluation mapping. This happens if $b \notin \mathbb{F}_{2^{m/2}} = V_{m/2}$ since m is a power of 2. This base isomorphism naturally extends to the polynomial rings over these fields:

$$\frac{\mathbb{F}_2[y]}{\langle \mu_b(y)\rangle}[x] \cong \mathbb{F}_{2^m}[x]\,, \tag{11}$$

where the variable y structurally behaves exactly as the element $b \in \mathbb{F}_{2^m}$. Because of this equivalence under the mapping $y \mapsto b$, the ideal generated by $s_l(x) - y$ is directly mapped to the ideal generated by $s_l(x) - b$. Consequently, we obtain the following quotient ring isomorphism:

$$\frac{\frac{\mathbb{F}_2[y]}{\langle \mu_b(y)\rangle}[x]}{\langle s_l(x) - y\rangle} \cong \frac{\mathbb{F}_{2^m}[x]}{\langle s_l(x) - b\rangle}\,. \tag{12}$$

Note that the LHS is also isomorphic to

$$\frac{\mathbb{F}_2[x,y]}{\langle s_l(x) - y, \mu_b(y)\rangle} \cong \frac{\mathbb{F}_2[x]}{\langle \mu_b(s_l(x))\rangle}\,. \tag{13}$$

In summary, we have an isomorphism $\frac{\mathbb{F}_2[x]}{\langle \mu_b(s_l(x))\rangle} \cong \mathbb{F}_{2^m}[x]/\langle s_l(x) - b\rangle$, that is clearly computed by reducing modulo $s_l(x) - b$. By picking $b = v_{m/2}$, we recover the usual `Encode` step of FAFFT, showing that $Q(x) = \mu_{v_{m/2}}(s_l(x))$.

To accelerate the reverse CRT map, it is beneficial to select an alternative b from a subspace larger than $V_{m/2+1}$, such that the resulting modulus $Q(x) = \mu_b(s_l(x))$ has a low Hamming weight. Subsection 4.3 provides a concrete example of searching for b to achieve a $Q(x)$ with the minimal Hamming weight.

4.2.2 Reverse CRT Map

The Reverse CRT map is required to combine the product obtained from the two smaller rings $\mathbb{F}_2[x]/\langle Q(x)\rangle$ and $\mathbb{F}_2[x]/\langle x^c\rangle$, in order to recover the full length product in the larger ring $\mathbb{F}_2[x]/\langle Q(x)x^c\rangle$.

Specifically, given the results $C_l \in \mathbb{F}_2/\langle Q(x)\rangle$ and $C_s \in \mathbb{F}_2[x]/\langle x^c\rangle$ from the smaller rings, we construct a polynomial $C \in \mathbb{F}_2[x]/\langle Q(x)x^c\rangle$ such that

$$\begin{aligned} C &\equiv C_l \pmod{Q(x)} \\ C &\equiv C_s \pmod{x^c}. \end{aligned}$$

To find the desired polynomial C, the reverse CRT map goes through two steps:

1. Determine the auxiliary polynomial $K \in \mathbb{F}_2[x]_{<c}$ by

$$K \equiv Q' \cdot (C_s - C_l) \pmod{x^c}, \tag{14}$$

   where Q' is the pre-computed polynomial given as $(Q^{-1} \bmod x^c)$.

2. From K, recover C as

$$C = C_l + K \cdot Q\,. \tag{15}$$

   Note that Q can be determined in advance. This allows us to exploit its sparsity and compute $K \cdot Q$ with simple shift-and-adds without secret-dependent branches or memory accesses.

4.3 Minimize Hamming Weight of Q

For optimization, we find $Q(x)$ with a low Hamming weight to reduce the computational cost of the CRT step by searching for a suitable constant a in the FAFFT. Observe that since $\mu_b(y) \in \mathbb{F}_2[y]$, if $b \in \mathbb{F}_{2^{32}} \setminus \mathbb{F}_{2^{16}}$ is a root, its Frobenius conjugates $b^2, b^{2^2}, \ldots, b^{2^{31}}$ are inherently roots of the same minimal polynomial. Consequently, the resulting modulus $Q(x) = \mu_b(s_l(x))$ depends only on the conjugacy class of b. With this, we were able to perform an exhaustive search efficiently as follows.

For each unvisited $b \in V_{21} \setminus V_{16}$, we compute $\mu_b(y)$, record the weight of $\mu_b(s_{11}(x))$, and finally mark all conjugates of b (i.e., $b^2, b^4, b^8, b^{16}, \ldots$) also as visited. This requires checking $(2^{21} - 2^{16})/32 = 63488$ b's in total. Note that for the 65536-point FAFFT in HQC-3 to work, we must have $b \in V_{m-l} = V_{21}$ where $m = 32$ and $l = \log_2(65536/m) = 11$.

In the end we picked $b = x_4x_2{+}x_3x_2x_0{+}x_3x_2{+}x_3x_1{+}x_3x_0{+}x_2x_1{+}x_2x_0{+}x_2{+}x_1 \in V_{21}\setminus V_{20}$, whose minimal polynomial $\mu = x^{32} + x^{26} + x^{25} + x^{24} + x^{22} + x^{20} + x^{19} + x^{14} + x^{12} + x^{11} + x^{10} + x^9 + x^4 + x + 1$ yields $\mathrm{wt}(\mu(s_{10}(x))) = 165$ and $\mathrm{wt}(\mu(s_{11}(x))) = 199$. Compared to a dense polynomial of the same degree which would have thousands of non-zero terms, this sparse structure drastically reduces the required XOR operations during the reverse CRT map.

4.4 Overview of the New Multiplication Algorithm

We describe the steps for combining the FAFFT and CRT techniques for polynomial multiplications in HQC. We use HQC-1 as a concrete example. In HQC-1, the polynomial length is $n = 17669 < 16384 + 1536$. The steps are listed as follows:

1. Lift to $\mathbb{F}_2[x]/\langle \mu(s_{10}(x))x^{3072}\rangle$.
2. Map to $\mathbb{F}_2[x]/\langle \mu(s_{10}(x))\rangle \times \mathbb{F}_2[x]/\langle x^{3072}\rangle$ by CRT: Using CRT, the polynomial in $\mathbb{F}_2[x]_{<32768+3072}$ is decomposed into two components.

3. Multiplication in $\mathbb{F}_2[x]/\langle Q(x)\rangle = \mathbb{F}_2[x]/\langle \mu(s_{10}(x))\rangle$: Perform the FAFFT multiplication in $\mathbb{F}_2[x]/\langle \mu(s_{10}(x))\rangle$ as described in Subsection 3.3.
4. Multiplication in $\mathbb{F}_2[x]/\langle x^{3072}\rangle$: Perform radix-16 Karatsuba in $\mathbb{F}_2[x]/\langle x^{3072}\rangle$ (refer to Subsection 6.1 for details).
5. Recover the product in $\mathbb{F}_2[x]/\langle \mu(s_{10}(x))x^{3072}\rangle$ via reverse CRT: Apply the reverse CRT mapping as described in Subsubsection 4.2.2.

We process HQC-3, which $n = 35851 < 32768 + 4096$, with similar methods as HQC-1. The two rings for CRT are $\mathbb{F}_2[x]/\langle \mu(s_{11})\rangle$ and $\mathbb{F}_2[x]/\langle x^{8192}\rangle$. The multiplication in the ring $\mathbb{F}_2[x]/\langle \mu(s_{11})\rangle$ is performed by FAFFT.

For HQC-5, the polynomial length is $n = 57637$, which is slightly less than $2^{16} = 65536$. Therefore, the multiplication can be processed by straightforward FAFFT multiplication.

Comparison with existing methods. Table 2 compares our approach with previous state-of-the-art FFT-based implementations for HQC-1. While standard FAFFT approaches [LJKH25] result in large polynomials with length $65,536$, hybrid methods attempt to reduce this size. Hybrid FAFFT [JLK+25] manages to reduce the transform size to $32,768$ by splitting the input polynomials into *two* partitions, which inherently requires applying *two* separate FAFFTs. Alternatively, [CCPY26] presented two approaches: Under their additive FFT+CRT framework, the algorithm inherently processes a larger data size, requiring a $65,536$-point transform. If adopting FAFFT to reduce the transform size, they must partition the polynomials within the same ring and fall back on Karatsuba for the cross terms. While this FAFFT+Karatsuba approach achieves a single $32,768$-point transform, it incurs significant computational overhead from the asymmetric cross-term multiplications (i.e., multiplying polynomials of length $1,285$ and $17,669$). In contrast, our FAFFT+CRT methodology extracts the optimal balance: it requires only one $32,768$-point FAFFT alongside two minor multiplications in the significantly smaller quotient ring $\mathbb{F}_2[x]/\langle x^{3072}\rangle$. This structural superiority avoids the massive cross-term overhead.

Table 2: Comparison of FFT-based multiplication strategies for HQC-1.

Implementation	Strategy	FFT Size	FFT Calls
[LJKH25]	Double-Size Standard FAFFT	65,536	1
[JLK+25]	Hybrid (FAFFT for 2/3 Karatsuba products)	32,768	2
[CCPY26]	FAFFT / Karatsuba for crossterms	32,768	1
[CCPY26]	Additive FFT + CRT	65,536	1
This work	FAFFT + CRT	32,768	1

5 Efficient Reed-Solomon Encoding and Decoding in HQC

This section details the Reed-Solomon (RS) coding algorithms utilized in our implementation. Unlike the Berlekamp-Massey algorithm favored by the HQC reference implementation [AAB+25], we adopt the Extended Euclidean Algorithm (EEA) to solve the key equation. The EEA-based approach is well-suited for SIMD-optimized implementations on the Cortex-M4 platform, as discussed in Subsection 6.2. A performance comparison between our optimized RS codec and the reference implementation is subsequently provided in Section 7.

5.1 Encoding as a Matrix-Vector Product

Following [AAB+25], the encoding process for a message vector $\mathbf{u} = (u_0, \ldots, u_{k-1})$ begins by representing it as a polynomial $u(x) = \sum_{i=0}^{k-1} u_i x^i$. The codeword polynomial is

computed as $c(x) = u(x) \cdot x^{n_1-k} + b(x)$, where $b(x) = x^{n_1-k}u(x) \bmod g(x)$ where $g(x)$ is the generator polynomial (see Subsubsection 2.1.1). This systematic construction ensures the original message appears in the k high degree coefficients.

To improve throughput over traditional polynomial long division, we perform the encoding as a matrix-vector product. We construct the generator matrix $[I_k \mid G]$, where the i-th row of G corresponds to the coefficients of $x^{n_1-k+i} \bmod g(x)$. The codeword vector is then computed as $\mathbf{c} = \mathbf{u} \cdot [I_k \mid G] = \mathbf{u} \parallel \mathbf{u} \cdot G$. Unlike the sequential nature of long division, this matrix-based approach facilitates vectorized implementation, which is critical in our M4 optimization (see Subsection 6.2).

5.2 Decoding Reed-Solomon Codes

5.2.1 RS Decoding Framework

The reference decoding procedure in [AAB+25] follows the Peterson-Gorenstein-Zierler (PGZ) framework [HP03]. Let the transmitted codeword $C(x) = \sum_{i=0}^{n_1-1} C_i x^i$ be defined over $\mathbb{F}_{256}$ with the primitive polynomial `0x11D`, where the parameters $[n_1, k]$ are specified in Subsubsection 2.1.1. Let α denote a primitive element of the field. The received polynomial is given by $R(x) = \sum_{i=0}^{n_1-1} R_i x^i = C(x) + e(x)$, where $e(x) = \sum_{i=0}^{\delta-1} e_i x^{j_i}$ represents the error polynomial with δ errors occurring at unknown locations j_i.

The decoding algorithm proceeds as follows:

1. Syndrome computation: Compute the 2δ syndromes $S_i = R(\alpha^i)$ for $i = 0, \ldots, 2\delta - 1$. Let $S(x) = \sum_{i=0}^{2\delta-1} S_i x^i$ be the syndrome polynomial.

2. Error locator polynomial: The error locator polynomial $\sigma(x) = 1 + \sum_{i=1}^{\delta} \sigma_i x^i = \sum_{i=0}^{\delta-1}(1 + \alpha^{j_i})x^i$ and the error evaluator polynomial $\omega(x)$ forms the key equation [HP03]:
$$\omega(x) \equiv \sigma(x)S(x) \pmod{x^{2\delta}}, \tag{16}$$
[AAB+25] uses the Berlekamp-Massey (BM) algorithm to solve the key equation for the error locator $\sigma(x)$.

3. Solving error locator: The roots of the error locator polynomial $\sigma(x)$ correspond to the error locations within the codeword. In [AAB+25], the additive FFT algorithm [GM10] is employed to evaluate $\sigma(x)$ over all elements of $\mathbb{F}_{256}$ to identify these roots. However, since the additive FFT evaluates polynomials over an additive affine subgroup, its efficiency is limited when finding roots that form a multiplicative group, as in the RS code setting of HQC. We accelerate this process by restricting the evaluation of $\sigma(x)$ only to the set of possible error locations (see Subsection 6.3 for related discussions).

4. Computing error value: Given the syndromes and solution of the error locator, Berlekamp [Ber68] first computed a polynomial $Z(x)$ defined as:
$$Z(x) = 1 + (S_1 + \sigma_1)x + \cdots + (S_w + \sigma_1 S_{w-1} + \cdots + \sigma_t)x^{\delta}. \tag{17}$$
Then the error values are calculated as:
$$e_{j_\ell} = \frac{Z(\beta_\ell^{-1})}{\prod_{\substack{i=1 \\ i \neq \ell}}^{\delta}(1 + \beta_i \beta_\ell^{-1})}, \tag{18}$$
where $\beta_i = \alpha^{j_i}$ are the inverses of the roots of $\sigma(x)$.

5. Correction: Subtract the computed error values from $R(x)$ to recover $C(x)$.

5.2.2 Extended Euclidean Algorithm-Based Decoding

Sugiyama et al. [SKHN75] proposed using the Extended Euclidean Algorithm (EEA) to solve the key equation (16). His decoding algorithm outputs two polynomials, $b_i(x)$ and $r_i(x)$, satisfying $\sigma(x) = \lambda b_i(x)$ and $\omega(x) = \lambda r_i(x)$ for some constant λ. After identifying the roots α^{-j_i} of $\sigma(x)$, error values e_i are determined using the Forney formula [For65]:

$$e_i = -\frac{\omega(\alpha^{-j_i})}{\sigma'(\alpha^{-j_i})} \quad . \tag{19}$$

The normalization constant λ is typically chosen such that $\sigma(0) = 1$, though it cancels out during Forney's formula computation and can be ignored in the core solver.

Constant-time implementation. To mitigate timing side-channel risks, we adopt the constant-time EEA variant (Algorithm 4) from [SY09], which decomposes the GCD procedure into a fixed number of iterations. By eliminating the highest-degree (2δ) coefficients in each step, the algorithm proceeds in constant-time execution. Its output consists of $G(x) = \lambda x^{\delta}\omega(x)$ and $R(x) = \lambda x^{\delta}\sigma(x)$, where δ is the number of errors. This specific output structure allows us to directly extract the coefficients of $\omega(x)$ and $\sigma(x)$ for the subsequent Forney formula evaluation.

Algorithm 4 Extended Euclidean algorithm for Reed-Solomon decoding in [SY09].

$d \leftarrow 0$
$F(x) \leftarrow x^{2\delta}$
$G(x) \leftarrow S(x)$
$V(x) \leftarrow 0$
$R(x) \leftarrow 1$
$i \leftarrow 1$
while $i \leq 2\delta$ **do**
    $d \leftarrow d - 1, G(x) \leftarrow xG(x), R(x) \leftarrow xR(x)$
    if $G_{2\delta} \neq 0$ & $d < 0$ **then**
        $d \leftarrow -d$ and **swaps** $G(x), R(x) \leftrightarrow F(x), V(x)$
    $G(x) \leftarrow F_{2\delta}G(x) - G_{2\delta}F(x)$
    $R(x) \leftarrow F_{2\delta}R(x) - G_{2\delta}V(x)$
    $i \leftarrow i + 1$

This algorithm is almost identical operationally to `safegcd` [BY19] specialized for solving the key equation. [BY19] has a *jump mode* leading to a lower asymptotic complexity and clear advantages in higher-degree polynomials (e.g., $n > 760$ in [JWYC24]), but even HQC-5 has an error locator of degree only 29. So we choose the more basic constant-time EEA from [SY09] because it is more straightforward to implement.

5.2.3 Comparative Analysis: EEA vs. Berlekamp-Massey

Although both approaches follow the PGZ framework, our EEA-based decoder introduces a shift in the computing structure compared to the conventional BM algorithm. This transition is essential for maximizing hardware utilization on the Cortex-M4.

Computational efficiency vs. regularity. Table 3 compares the $\mathbb{F}_{256}$ operation counts for the two algorithms. While the BM algorithm yields a lower operation count, it only produces the error locator polynomial $\sigma(x)$. In contrast, the EEA simultaneously generates both $\sigma(x)$ and the error evaluator $\omega(x)$. More importantly, the EEA exhibits a better data access pattern for parallel architectures. As shown in Algorithm 4, the EEA relies only on vector-scalar multiplications driven by coefficients of the highest degree. Conversely, the BM algorithm requires an *inner product* between the syndrome polynomial

and the internal state. These inner product operations are inherently sequential and pose significant bottlenecks for SIMD optimization.

Error value computation. The EEA approach further simplifies error evaluation by utilizing Forney's formula (19), which reduces the problem to parallel polynomial evaluations via Horner's rule. This regularity is highly conducive to vectorized processing. In contrast, the BM-based method [AAB+25] requires the intermediate computation of a $Z(x)$ polynomial through irregular patterns (17), followed by a denominator computation (18) that poses challenges for parallelization.

SIMD optimization and experimental results. Consequently, while the EEA incurs a higher operation count, its regular execution flow and SIMD-friendly primitives enable a more efficient implementation. The powerful SIMD optimizations for $\mathbb{F}_{256}$ arithmetic in Subsection 6.2 convert the algorithmic regularity of the EEA into substantial performance gains. Our experiments in Subsection 7.2 confirm these results, demonstrating that the EEA-based decoder outperforms the BM-based implementation.

Table 3: Number of $\mathbb{F}_{256}$ operation for solving the key equation in HQC.

	HQC-1		HQC-3		HQC-5	
Approach	Mul.	Inv.	Mul.	Inv.	Mul.	Inv.
Berlekamp-Messay	1,695	60	1,928	64	6,322	116
Extended Euclidean algorithm Algorithm 4	3,285	15	3,727	16	12,035	29

6 Implementation on Arm Cortex-M4

In this section, we present our optimized implementations on ARM Cortex-M4 for two fundamental computational structures. Subsection 6.1 describes our strategy in polynomial ring arithmetic. Subsection 6.2 analyzes arithmetic over $\mathbb{F}_{256}$, implementing and comparing multiplication approaches. Subsection 6.3 shows how the Reed-Solomon codes can be accelerated with these operations.

6.1 Boolean Polynomial Multiplication

Our new multiplication algorithm illustrated in Subsection 4.4 uses the CRT to decompose the polynomial in $\mathbb{F}_2[x]_{<32768+3072}$ and $\mathbb{F}_2[x]_{<65536+8192}$ into $\mathbb{F}_2[x]/\langle\mu(s_{10})\rangle \times \mathbb{F}_2[x]/\langle x^{3072}\rangle$ and $\mathbb{F}_2[x]/\langle\mu(s_{11})\rangle \times \mathbb{F}_2[x]/\langle x^{8192}\rangle$ in HQC-1 and 3, respectively. Multiplications in rings $\mathbb{F}_2[x]/\langle\mu(s_{10})\rangle$ and $\mathbb{F}_2[x]/\langle\mu(s_{11})\rangle$ are performed by the FAFFT algorithm in Subsubsection 6.1.2, and multiplications in rings $\mathbb{F}_2[x]/\langle x^{3072}\rangle$ and $\mathbb{F}_2[x]/\langle x^{8192}\rangle$ are processed with the *truncated Karatsuba* algorithm in radix-16 representation.

6.1.1 Radix-16 Polynomial Multiplication In $\mathbb{F}_2[x]/\langle x^c\rangle$

FP register cache for 64-bit base multiplication. To establish efficient multiplications, we first optimize the base multiplication by overcoming the register limitations on the Cortex-M4. Subsection 2.3 introduced the 32-bit $\mathbb{F}_2[x]$ radix-16 multiplication as the basic building block. However, while building longer multiplication, e.g., 64-bit multiplication, the overhead of memory access will inevitably occur due to limited register numbers. To address this, we implement 64-bit multiplications with the same strategy as in [JLK+25] and floating-point(FP) registers. By utilizing the FP registers as cache, we thus eliminate numerous slow memory access instructions, e.g., `ldr`/`str` and calling conventions between function calls of smaller base multiplication.

32-bit truncated multiplication. The 32-bit truncated multiplication is a core building block for modular ring arithmetic, computing only the lower 32 bits of a 32-bit base multiplication. Our optimization focuses on identifying an accumulation sequence that minimizes normalization steps (masking with `0x11111111`). Since radix-16 segments can only accommodate values up to 15, any intermediate term exceeding 16 would cause overflow and lead to incorrect results.

Figure 3 illustrates our optimized sequence. In this truncated form, we retain only the lower 32 coefficients (the first four segments). For the highest segment of the result, carry propagation to bits of higher degree is safely ignored. This allows for the accumulation of one upper half and two lower halves without intermediate masking. Compared to the 32-bit base multiplication, which consumes 50 instructions, the 32-bit truncated multiplication significantly reduces the cost to only 30 instructions.

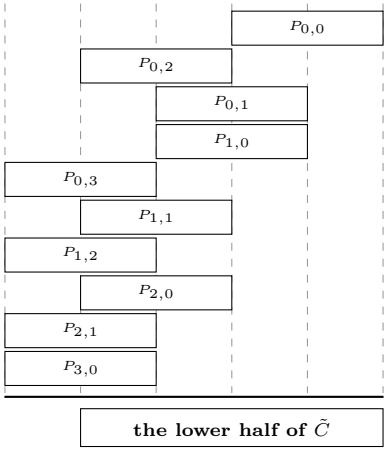

Figure 3: Constructing the 32-bit truncated multiplication.

Truncated Karatsuba. The truncated Karatsuba algorithm significantly reduces computational overhead by omitting higher-order terms unnecessary for modular reduction. For n-degree polynomials $A = A_1 x^{\frac{n}{2}} + A_0$ and $B = B_1 x^{\frac{n}{2}} + B_0$, the truncated product $\mathrm{TM}_n(A, B)$ is defined recursively as:

$$\mathrm{TM}_n(A, B) = A_0 B_0 + \left(\mathrm{TM}_{\frac{n}{2}}(A_0, B_1) + \mathrm{TM}_{\frac{n}{2}}(A_1, B_0)\right) x^{\frac{n}{2}} \quad . \tag{20}$$

Unlike the standard Karatsuba approach, which requires three full $n/2$-degree multiplications, this variant involves only one full multiplication ($A_0 B_0$) and two truncated multiplications of half the degree, continuing until the base case of $n = 32$ is reached.

6.1.2 The FAFFT Multiplication in $\mathbb{F}_2[x]/\langle \mu(s_{10}) \rangle$ and $\mathbb{F}_2[x]/\langle \mu(s_{11}) \rangle$

Skipping zeros. In the HQC-1 and HQC-3 parameter sets, the input polynomials are padded to 16,384+2,048 and 32,768+4,096 bits, respectively. Since these lengths are not powers of two, directly applying `BasisCvt` (Algorithm 2) would involve redundant operations on zero coefficients. To optimize this, we skip these redundant computations by performing an explicit division of the input polynomials by s_{14} (for HQC-1) or s_{15} (for HQC-3).

Due to the structure of the vanishing polynomial s_i, where non-zero coefficients only appear at degrees of 2^j, this division simplifies to `XOR`ing the higher-degree bits (2,048 or 4,096 bits) into the lower-degree bits (16,384 or 32,768 bits) with specific power-of-two shifts. After this initial division, the remaining steps of `BasisCvt` are performed as standard 2-powered length transformations according to Algorithm 2.

Optimizing `XOR` sequences for constant multiplications. The `Butterfly` operation (Algorithm 3) and the FAFFT encoding process involve multiplications by predefined constants $s_i(b)$ and $X_j(b)$ from (10), respectively. Since these constants are fixed in $\mathbb{F}_{2^{32}}$, each multiplication can be modeled as a matrix-vector product over $\mathbb{F}_2$, analogous to the linear layers in symmetric ciphers. To maximize throughput, we utilize optimization tools from [XZL+20, LJKH25] to generate highly efficient `XOR` sequences. This approach minimizes the total number of `XOR` instructions required to implement these constant multiplications, significantly reducing the computational overhead of the FAFFT stages.

6.2 Optimized $\mathbb{F}_{256}$ Arithmetic on Cortex-M4

The performance of RS coding in HQC is dominated by finite field multiplication over $\mathbb{F}_{256}$. In this section, we review the state-of-the-art byte-sliced implementation and propose a novel approach based on the radix-16 representation to accelerate variable-variable multiplications.

`0x11D`

Byte-sliced multiplication. The byte-sliced technique [BCH+23] is highly efficient for *vector-scalar* multiplication. By packing four 8-bit elements into a 32-bit register, it processes four multiplications against a single scalar simultaneously.

While the initial cost for a 4×1 multiplication is 68 cycles (17 cycles/multiplication), the cost is significantly amortized when the scalar is reused across a large vector. For a vector of size $4n$, the instruction count is approximately $44 + 24n$, reaching an optimal throughput of roughly 6 cycles/multiplication.

Algorithm 5 $\mathbb{F}_{256}$ element-wise multiplication in radix-16 representation.

Input:
A: 32-bit integer packing four elements (a_3, a_2, a_1, a_0) in radix-16 representation
B: 32-bit integer packing four elements (b_3, b_2, b_1, b_0) in radix-16 representation
pconst=`0x00011101` (`0x1d` in radix-16 representation); *mconst*=`0x11111111`
Output: C: 32-bit integer packing results (c_3, c_2, c_1, c_0), where $c_i = a_i \times b_i \in \mathbb{F}_{256}$

1: **for** element $i = 0, \ldots, 3$ **do**
2: **and** a_i, *mconst*, A, lsr#k_i ▷ extract a_i from A, $k_0 = 0,\ k_1 = 2,\ k_2 = 1,\ k_3 = 3$
3: **and** b_i, *mconst*, B, lsr#k_i ▷ extract b_i from B
4: **umull** t_i, t_4, a_i, b_i ▷ $c_i = [t_i, t_4] = a_i \times b_i$
5: **and** t_4, t_4, *mconst* ▷ t_4 is the part of degree 8 to 14 of c_i
6: **umull** t_5, t_4, t_4, *pconst* ▷ reduce 16-bit c_i to 8-bit c_i
7: **mla** t_5, t_4, *pconst*, t_5 ▷ multiply t_4 without masking *mconst*
8: **eor** t_i, t_i, t_5
9: **and** t_i, t_i, *mconst*
10: **orr** C, t_0, t_1, lsl#2 ▷ pack 4 results into one register
11: **orr** C, C, t_2, lsl#1
12: **orr** C, C, t_3, lsl#3

Radix-16 element-wise multiplication. While byte-slicing excels in vector-scalar scenarios where one operand is fixed, it becomes suboptimal for multiplications where both operands are dynamic. To overcome this drawback of byte-slicing, we propose a vector-vector element-wise multiplication leveraging the radix-16 representation (Subsection 2.3). This approach, detailed in Algorithm 5, maintains computational efficiency even when both multipliers vary at runtime.

Algorithm 5 uses one `umull` instruction for multiplying 8-bit polynomials to one 16-bit product. The reduction of intermediate terms of degree 8 to 14 is processed by one `umull` and one `mla` instruction. It costs 35 instructions for a 4×4 element-wise multiplication, or approximately 8.75 cycles/multiplication. We note that the cost of data conversion is 3

per multiplication (12 cycles for converting 4 elements, see Subsection 2.3).

Constant matrix-vector multiplication with lazy reduction. The primary advantage of the radix-16 representation lies in *constant matrix-vector multiplication*, shown in Algorithm 6. This scenario allows for two major optimizations: 1) Offline Precomputation: The radix-16 form of the constant matrix is computed at compile time. 2) Lazy Reduction: Multiple intermediate products can be accumulated before performing reduction.

In Algorithm 6, a 4×16 matrix-vector multiplication costs 291 cycles (excluding conversion), or 4.55 cycles/multiplication. For larger matrices, such as the 32×24 matrix in HQC-3, the performance improves to 4.45 cycles/multiplication. Generalizing to arbitrary dimensions, performing a matrix-vector multiplication with a constant $4m \times 4n$ matrix and a $4n$-element variable vector incurs a cost of $(19 \times 4n - 4 \times (1 + 2n) + 20 + 3)m = (68n + 19)m$ cycles, or optimally $68/16 = 4.25$ cycles/multiplication. Similarly, we note that the cost of data conversion is 3 cycles per element.

Algorithm 6 $\mathbb{F}_{256}$ matrix-vector multiplication in radix-16 representation.

Input:
vector: a vector of 16 elements in FP registers $\{s_0 - s_{15}\}$
matrix: a pointer to a pre-computed 4×16 matrix in radix-16 representation
pconst=`0x00011101` (`0x1d` in radix-16 representation); *mconst*=`0x11111111`
Output: *result*: the product vector of length 4 in radix-16 representation

1: **for** $i = 0, \ldots, 16 - 1$ **do** ▷ 16 is the length of the vector
2: **ldr** t_9, *matrix*[i] ▷ load 4 matrix elements
3: **vmov** t_{10}, s_i ▷ extract one element in the input vector
4: **for** $j = 0, \ldots, 3$ **do**
5: **and** t_8, *mconst*, t_9, lsr#k_p ▷ $k_0 = 0,\ k_1 = 2,\ k_2 = 1,\ k_3 = 3$
6: **umlal** t_{2j}, t_{2j+1}, t_8, t_{10} ▷ use **umull** when $i = j = 0$
7: **and** t_{2j}, t_{2j}, *mconst* ▷ skip when $i = 15$
8: **and** t_{2j+1}, t_{2j+1}, *mconst* ▷ skip when i is even
9: **for** element $j = 0, \ldots, 3$ **do** ▷ reduction and accumulation are in different loops
10: **umull** t_8, t_9, t_{2j+1}, *pconst*
11: **mla** t_8, t_9, *pconst* , t_8
12: **eor** t_{2j}, t_{2j}, t_8
13: **and** t_{2j}, t_{2j}, *mconst*
14: **orr** *result*, t_0, t_2, lsl#2 ▷ pack results
15: **orr** *result*, *result*, t_4, lsl#1
16: **orr** *result*, *result*, t_6, lsl#3

6.3 Application to RS Codec in HQC

In our HQC Reed-Solomon codec implementation, we strategically deploy two $\mathbb{F}_{256}$ multiplication techniques —byte-slice and radix-16— to maximize throughput based on specific data access patterns. While byte-slicing serves as the default for general operations, we switch to radix-16 for specific high-performance scenarios.

Specifically, in RS encoding and syndrome computation, which involve matrix-vector products with predefined matrices, the radix-16 representation is particularly advantageous for its efficient accumulation of intermediate products with the lazy reduction technique.

Furthermore, during the evaluation of the error evaluator polynomial $\omega(x)$ and the derivative of the locator polynomial $\sigma'(x)$ via Forney's formula, the operation requires evaluating these polynomials over a set of error positions $\{\alpha^{-j_i}\}$, which are inverses of the roots of $\sigma(x)$. When applying Horner's rule to multiple α^{-j_i} simultaneously, the computation shifts from a vector-scalar pattern to a vector-vector element-wise scenario. Our radix-16

implementation excels in this parallel evaluation context, making it a superior choice over the byte-slice approach for these performance-critical decoding stages.

7 Performance Evaluation

7.1 Experimental Setup

We utilize the STM32L4R5-Nucleo-144 development board as our target platform, aligning with the hardware used in the `pqm4` benchmarking framework. The board features the STM32L4R5ZIT6 microcontroller, based on the ARM Cortex-M4 architecture equipped with a floating-point unit (FPU). The microcontroller includes 640 KB of SRAM and 2 MB of flash memory, capable of operating at clock frequencies up to 120 MHz.

For benchmarking, we adopt the `pqm4` framework (version `a24bb4b`). Following standard practice for this framework, we benchmark all implementations at a clock frequency of 20 MHz to ensure zero wait states when accessing instructions or data from flash memory. The code is compiled using `arm-none-eabi-gcc` version 10.3.1 with optimization flags set to `-O3` (default in `pqm4`).

Empirical time constancy. In extensive benchmarking, measured cycle counts for benchmarked operations remained strictly identical across all executions with uniformly random inputs.

7.2 Component-level Benchmarks

Table 4 presents the cycle counts for polynomial multiplication in the ring $\mathcal{R} = \mathbb{F}_2[X]/\langle X^n + 1\rangle$. Our implementation achieves speedups of 18.3% and 15.1% for HQC-1 and HQC-3, respectively, compared to previous implementations. These results indicate that the FFT-based approach combined with CRT outperforms Karatsuba methods at all security levels.

This finding aligns with recent studies in [CCPY26], which suggest that FFT-based methods are more efficient than Toom-Karatsuba ones on platforms without long carry-less multiplication instructions. Our work further validates that a tailored integration of FAFFT and CRT yields strong performance improvements on the Cortex-M4 for polynomials of length $n = 2^i + c$ at smaller degrees.

Table 4: Cycle count for ring ($\mathcal{R}$) multiplication in HQC.

Implementation	HQC-1	HQC-3	HQC-5
[LJKH25]	2 841 792	6 279 473	6 290 004
[JLK+25] radix-16	2 148 305	6 083 951	12 457 616
[JLK+25] hybrid fafft	2 521 568	5 697 430	NA
This work	1 753 860	4 836 147	6 290 004

Table 5 summarizes the performance of the error correction components. For encoding, our implementation leverages an optimized matrix-vector product, reducing the computation time by approximately 90% compared to the reference implementation. For decoding, we achieve reductions in cycle counts ranging from 36.4% to 45.2% across different security levels. These benchmarks suggest that a Sugiyama decoder based on the Extended Euclidean Algorithm (EEA) is highly efficient for Reed-Solomon codes, even on architectures without vector instructions. We anticipate that these advantages would be even more significant on platforms with native vector SIMD support.

Table 5: Cycle count of the $\mathcal{C}$.Encode/$\mathcal{C}$.Decode operation in HQC.

Implementation	$\mathcal{C}$.Encode			$\mathcal{C}$.Decode		
	HQC-1	HQC-3	HQC-5	HQC-1	HQC-3	HQC-5
PQClean [KSSW22]	87 743	140 969	331 044	1 507 904	2 120 793	4 262 364
This work	7945	14 878	26 752	909 832	1 348 110	2 337 350

7.3 Overall Performance on ARM Cortex-M4

Table 6 compares the performance of our work against the reference implementation (PQClean [KSSW22]) and recent state-of-the-art optimizations [LJKH25, JLK+25]. Our work achieves significant performance gains across the primary security levels. For HQC-1, we report speedups of 19.5% and 20.4% for Encapsulation and Decapsulation, respectively, compared with the best previous results (radix-16 [JLK+25]). For HQC-3, the improvements are 15.9% and 15.9% compared with the hybrid FAFFT implementation.

These gains stem from three main contributions: the improved polynomial multiplication, the optimized RS decoder, and the incorporation of the common input transform proposed in [CCPY26], which eliminates one FAFFT transform during encryption.

Notably, for HQC-5, we retain the polynomial multiplication implementation from [LJKH25, JLK+25], as our FAFFT+CRT technique is inapplicable to this parameter set. Consequently, the performance improvements observed in HQC-5 over the state-of-the-art [JLK+25] stem entirely from our codec engineering and transform caching, serving as a natural ablation study. Specifically, the 9.1% speedup in encapsulation is driven strictly by transform caching, which saves approximately one-third of a standard multiplication's cost. This clearly illustrates the difference from [JLK+25], which employs the identical multiplication but lacks the transform caching mechanism. In contrast, the 10.5% speedup in decapsulation is a combined result of both transform caching and our new SIMD-friendly EEA decoder. As shown in Table 6, the absolute cycle reduction for decapsulation is significantly larger than that for encapsulation. This pronounced difference clearly isolates the standalone impact of the EEA-based decoder, proving that our codec optimizations deliver substantial and independent performance gains.

Table 6: Kilo-cycle counts of HQC implementations on ARM Cortex-M4.

Implementation	Kilo-Cycle Count									Stack†
	HQC-1			HQC-3			HQC-5			
	Key	Enc	Dec	Key	Enc	Dec	Key	Enc	Dec	(B)
PQClean [KSSW22]	14 156	28 566	43 938	42 819	85 942	130 314	82 275	164 827	250 419	56 036
[LJKH25]	4016	7835	13 082	9497	18 328	28 993	12 700	24 082	38 887	62 856
[JLK+25] radix-16	3322	6448	11 001	9302	17 937	28 407	18 868	36 417	57 390	30 004
[JLK+25] hybrid fafft	3696	7195	12 121	8915	17 164	27 247	–	–	–	56 736
This work	2928	5188	8749	8055	14 427	22 877	12 702	21 890	34 772	59 664

† Stack usage (in bytes) of the decapulation operation on HQC-1.

7.3.1 Resource Utilization and Energy Estimation

In addition to cycle counts, we also analyze the resource utilization of our implementation, including Flash (storing code and constant tables) and RAM usage. Table 7 details the memory footprint of our work. The peak stack and Flash usages are approximately 163 KB and 416 KB, respectively. Both fit well within the 640 KB SRAM and 2 MB Flash memory available on the STM32L4R5 MCU, leaving ample space for other applications.

When compared to the implementations in Table 6, the memory footprints clearly diverge based on the underlying multiplication strategies. Methods utilizing the radix-16 Karatsuba approach intrinsically require less stack space. In contrast, all FFT-based algorithms

Table 7: Memory footprint (in bytes) of the proposed HQC implementation.

Parameter	Flash(code + data)	RAM(data+bss)	Stack		
			Keygen	Encap	Decap
HQC-1	269 328	2836	42 544	52 864	59 664
HQC-3	306 216	2836	84 512	105 736	119 352
HQC-5	416 120	2836	111 704	140 976	162 760

naturally exhibit a larger memory footprint due to the necessity of storing intermediate transform evaluations. Our stack usage aligns strictly with this expected algorithmic characteristic. Given the abundant SRAM provided by modern embedded platforms, this inherent architectural design presents no barrier to practical deployment.

For a conservative engergy estimation, we adopt the standard LDO run mode profile (110 μA/MHz) and the default 3.3V supply of the STM32L4R5 Nucleo-144 board, as specified in the datasheet [STM20]. Operating at a fixed frequency of 20 MHz with zero wait states, the baseline power consumption is $\sim$ 7.26 mW (110 μA/MHz $\times$ 20 MHz $\times$ 3.3V). The energy per operation is linearly projected by multiplying this power by the execution time (Cycle Count/20 MHz). This projection yields an estimated energy consumption of $\sim$ 1.06/1.88/3.18 mJ for KeyGen/Encap/Decap operations in our HQC-1 implementation.

7.4 Remarks on Portability

We provide both standard C and Cortex-M4 assembly implementations to ensure portability. Porting to other ARM variants or embedded RISC-V [RIS26] architectures is straightforward, though the resulting performance benefits will naturally depend on the available instruction set extensions of the target platform.

RISC-V. The FAFFT core relies on fundamental bitwise operations, making its transition to RISC-V seamless and expected to yield similar improvements on baseline integer ISAs. However, the efficiency of our radix-16 arithmetic depends heavily on specific extensions. On platforms with only the base M-extension, the lack of native multiply-and-accumulate instructions might slightly reduce radix-16 efficiency compared to the Cortex-M4. Conversely, the Zbc extension introduces native carry-less multiplication (clmul), which [CGKT22] demonstrated to outperform radix-16 approaches. Leveraging the Zbc extension thus presents an avenue for even greater acceleration in both polynomial multiplication and RS decoding on RISC-V.

Platforms with vector instructions. Our reduction in algorithmic operation count inherently benefits high-end platforms equipped with SIMD extensions (e.g., x86 or ARM Cortex-A series). Nevertheless, since modern vector ISAs natively support carry-less multiplication, achieving peak performance on these platforms would require redesigning the FFT-core to directly exploit these instructions, similar to the SIMD-oriented approach proposed in [CCPY26].

Acknowledgements

Academia Sinica authors were supported by Academia Sinica Grand Challenge Project (AS-GCP-114-M01), and National Science and Technology Council grants 114-2221-E-001-014-MY3, 113-2221-E-001-023-MY3, 114-2634-F-001-001-MBK.

References

[AAB+22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[AAB+25] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, Pascal Veron, Paulo L. Barreto, Santosh Ghosh, Shay Gueron, Tim Güneysu, Rafael Misoczki, Jan Richter-Brokmann, Nicolas Sendrier, Jean-Pierre Tillich, and Valentin Vasseur. HQC. Technical report, pqc-hqc.org, 2025. available at https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf.

[AGZ] Nicolas Aragon, Philippe Gaborit, and Gilles Zémor. Hqc-rmrs, an instantiation of the hqc encryption framework with a more efficient auxiliary error-correcting code. https://arxiv.org/abs/2005.10741.

[Arm20] Arm Limited, Cambridge, UK. *Arm Cortex-M4 Processor Technical Reference Manual*, issue 1.0 edition, September 2020. Document number 100166_0001_04_en, available at https://documentation-service.arm.com/static/5fce431be167456a35b36ade.

[BCH+23] Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J. Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. Oil and vinegar: Modern parameters and implementations. *IACR TCHES*, 2023(3):321–365, 2023.

[Ber68] Elwyn Berlekamp. *Algebraic coding theory*. World Scientific, USA, 1968.

[Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 260–272, Haifa, Israel, January 20–22, 1997. Springer Berlin Heidelberg, Germany.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR TCHES*, 2019(3):340–398, 2019.

[Can89] David G Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285–300, 1989.

[CC21] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. *IACR TCHES*, 2021(3):125–148, 2021.

[CCK+17] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster multiplication for long binary polynomials. *CoRR*, abs/1708.09746, 2017.

[CCK+18] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Multiplying boolean polynomials with Frobenius partitions in additive fast fourier transform. *CoRR*, abs/1803.11301, 2018.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the intel haswell and ARM Cortex-M4. *IACR TCHES*, 2021(3):97–124, 2021.

[CCPY26] Ming-Shing Chen, Chun-Ming Chiu, Chun-Tao Peng, and Bo-Yin Yang. Accelerating HQC with additive FFT. Cryptology ePrint Archive, Report 2026/14, 2026. To appear in IACR TCHES 2026(2).

[CGKT22] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-less to BIKE faster. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 2022*, volume 13269 of *LNCS*, pages 833–852, Rome, Italy, June 20–23, 2022. Springer, Cham, Switzerland.

[DF03] D.S. Dummit and R.M. Foote. *Abstract Algebra*. Wiley, 2003.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554, Santa Barbara, CA, USA, August 15–19, 1999. Springer Berlin Heidelberg, Germany.

[For65] George Forney. On decoding bch codes. *IEEE Transactions on Information Theory*, 11:549–557, 1965.

[GM10] Shuhong Gao and Todd D. Mateer. Additive fast fourier transforms over finite fields. *IEEE Trans. Inf. Theory*, 56(12):6265–6272, 2010.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 341–371, Cham, 2017. Springer International Publishing.

[HP03] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[JLK+25] Jihoon Jang, Myeonghoon Lee, Donggeun Kwon, Seokhie Hong, Suhri Kim, and Sangjin Lee. Efficient polynomial multiplication for HQC on ARM cortex-M4. Cryptology ePrint Archive, Report 2025/1939, 2025.

[JWYC24] Li-Jie Jian, Ting-Yuan Wang, Bo-Yin Yang, and Ming-Shing Chen. Jumping for bernstein-yang inversion. In Tianqing Zhu and Yannan Li, editors, *ACISP 24, Part II*, volume 14896 of *LNCS*, pages 103–123, Sydney, NSW, Australia, July 15–17, 2024. Springer, Singapore, Singapore.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. Cryptology ePrint Archive, Report 2022/337, 2022.

[LANHC16] Sian-Jheng Lin, Tareq Y. Al-Naffouri, Yunghsiang S. Han, and Wei-Ho Chung. Novel polynomial basis with Fast Fourier Transform and its application to Reed–Solomon erasure codes. *IEEE Transactions on Information Theory*, 62(11):6284–6299, 2016.

[LC04] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., USA, 2004.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed-solomon erasure codes. In *55th FOCS*, pages 316–325, Philadelphia, PA, USA, October 18–21, 2014. IEEE Computer Society Press.

[LCK+18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast Fourier transform. In Manuel Kauers, Alexey Ovchinnikov, and Éric Schost, editors, *Proceedings of the 2018 ACM*

on International Symposium on Symbolic and Algebraic Computation, ISSAC 2018, New York, NY, USA, July 16-19, 2018, pages 263–270. ACM, 2018.

[LJKH25] Myeonghoon Lee, Jihoon Jang, Suhri Kim, and Seokhie Hong. Improved Frobenius FFT for code-based cryptography on Cortex-M4. *IEEE Internet of Things Journal*, 12(17):36318–36327, 2025.

[MPR+24] Dustin Moody, Ray Perlner, Andrew Regenscheid, Angela Robinson, and David Cooper. Transition to post-quantum cryptography standards. NIST Internal Report (IR) NIST IR 8547, National Institute of Standards and Technology, Gaithersburg, MD, 2024. https://doi.org/10.6028/NIST.IR.8547.ipd.

[RIS26] RISC-V International. *The RISC-V Instruction Set Manual*. RISC-V International, 2026. Document Version 20260323 (25b3fbc). Available at https://github.com/riscv/riscv-isa-manual.

[RLC+25] Antonio Ras, Antoine Loiseau, Mikaël Carmona, Simon Pontié, Guénaël Renault, Benjamin Smith, and Emanuele Valea. PHOENIX: Crypto-agile hardware sharing for ML-KEM and HQC. Cryptology ePrint Archive, Report 2025/601, 2025.

[SKHN75] Yasuo Sugiyama, Masao Kasahara, Shigeichi Hirasawa, and Toshihiko Namekawa. A method for solving key equation for decoding goppa codes. *Information and Control*, 27:87–99, 1975.

[STM20] STMicroelectronics. Ds12023 rev5: Datasheet - stm32l4s5xx stm32l4s7xx stm32l4s9xx, 2020. https://www.st.com/resource/en/datasheet/stm32l4r5vi.pdf.

[SY09] Dilip V. Sarwate and Zhiyuan Yan. Modified euclidean algorithms for decoding reed-solomon codes. *Proceedings of the 2009 IEEE International Symposium on Information Theory*, 2:1398 – 1402, 2009.

[vdHL17] Joris van der Hoeven and Robin Larrieu. The Frobenius FFT. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, page 437–444, New York, NY, USA, 2017. Association for Computing Machinery.

[XZL+20] Zejun Xiang, Xiangyong Zeng, Da Lin, Zhenzhen Bao, and Shasha Zhang. Optimizing implementations of linear layers. *IACR Trans. Symm. Cryptol.*, 2020(2):120–145, 2020.