Accelerating HQC with Additive FFT

Ming-Shing Chen[1], Chun-Ming Chiu[1], Chun-Tao Peng[1] and Bo-Yin Yang[1]

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

Abstract. This paper presents an accelerated implementation of the Hamming Quasi-Cyclic (HQC) key encapsulation mechanism by leveraging additive Fast Fourier Transform (FFT) for polynomial multiplication (*polymuls*). A common challenge when applying FFT-based polymuls to HQC are the polynomial degrees fractionally greater than powers of two, making standard FFT polymuls inefficient for those parameters. We introduce a novel combination of additive FFT with the Chinese Remainder Theorem (CRT) to multiply such just-above-power-of-two degree polynomials.
Further optimizations are achieved by caching the FFT transforms of the public and secret keys during or after key generation and reusing the transform of the shared random polynomial during the encapsulation and decapsulation processes. This approach significantly reduces redundant computations.
The effectiveness of these optimizations is evaluated across various hardware platforms, including x86-64 with AVX2 and Galois Field New Instructions (GFNI), as well as ARM NEON on Apple M1 and Cortex-A72 processors. Benchmarks show that on platforms with long carry-less multiplication instructions like `PCLMULQDQ`, the proposed caching and reuse strategies allow FFT-based implementations to outpace traditional Toom-Karatsuba methods in the Encap and Decap operations of the HQC scheme, even with comparable raw multiplication speeds. On platforms lacking long carry-less multiplication instructions, the additive FFT approach already gives the superior performance.

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

1 Introduction

To counter the threat of quantum computers, the NIST Post-Quantum Cryptography (PQC) Standardization Process [MPR+24] has been ongoing since 2016. Among the selected finalists, *Hamming Quasi-Cyclic (HQC)* [AAB+22, AAB+25] stands out as a code-based public key encapsulation mechanism (KEM) rooted in the hardness of decoding quasi-cyclic codes—a problem believed to be resistant even against quantum adversaries. Recently, HQC has been officially selected for standardization, becoming the future standard KEMs.

One of the most performance-critical operations in HQC is implementing multiplication in the polynomial ring $\mathcal{R} = \mathbb{F}_2[x]/(X^n + 1)$. The multiplication length for HQC ranges from 18k to 58k bits, depending on the chosen security level. Efficient multiplication in this ring is thus essential for optimizing key generation, encapsulation, and decapsulation times.

The reference and optimized implementations in[AAB+22, AAB+25] uses Toom and Karatsuba for polynomial multiplications (polymuls). Given the polynomial lengths and previous studies in code-based cryptosystems, we believe that Fast Fourier Transform (FFT)-based algorithms can enhance the performance of polymuls used in HQC. In this paper, we explore the use of FFT-based algorithms to replace the original polymuls in [AAB+22, AAB+25]. We evaluate our approach across different parameter sets and CPUs,

including x86 with AVX2 and GFNI instructions, Raspberry Pi4 and Mac M1.

1.1 Our Contributions

In this work, we present optimized implementations of HQC on both Arm NEON and x86 AVX2 platforms, including support for the Galois Field New Instructions (GFNI) [Int21] as well as CPUs lacking long $\mathbb{F}_2$-polymul instructions.

We propose a novel combination of additive FFT with the Chinese Remainder Theorem (CRT) to multiply in HQC-1 and HQC-3. This approach allows us to multiply polynomials of degree slightly greater than powers of 2, where direct usage of additive FFT is inefficient.

FFT-based multiplications are particularly beneficial in the **Encryption** function. This function includes two polynomial multiplications that share a common input polynomial. We eliminate redundant FFT transforms, saving roughly 1/8 computation time on x86 AVX2 and 1/6 computation time on ARM NEON for two FFT multiplications.

We also introduce a caching strategy for caching the (FFT) transformed polynomials in the encapsulation and decapsulation keys, These cached transforms are reused in the encapsulation and decapsulation phases, further reducing computational overhead.

Finally we present time-constant optimized encoding and decoding of the Reed-Muller and Reed-Solomon codes used in HQC that mitigate side-channel vulnerabilities.

Source code The source code for all our implementations is available under CC0 copyright-waiver at https://github.com/ChunTaoPengim/HQC_with_addFFT_tches2026.

1.2 Related Works

HQC optimization is a persistent focus of hardware research. Work began with [DXN+24] in 2023, which focused on accelerating sparse polynomial multiplication, and followed by [ABP25], which presented a fully parameterized RTL implementation. Recently, research has concentrated on FFT and system integration: Ras et al. [RLC+25b] proposed sharing addFFT units for area/throughput gains. This group subsequently [RLC+25a] presented an optimized FFT implementation integrated into a RISC-V System-On-Chip.

The two forms of additive FFT used in our implementation have been studied in the literature. Chen et al. [CCK+17] applied additive FFT with Kronecker segmentation to multiplication of long binary polynomials. [LCK+18, CCK+18] applied Frobenius additive FFT to $\mathbb{F}_2[x]$ multiplications, showing that even with dedicated $\mathbb{F}_2$-polymul instructions, FFT outperforms Karatsuba when multiplying polynomials that are $10k$'s-bits long.

The polymul performance in schemes like BIKE and HQC on embedded systems is a highly competitive and active area of research. Early FFT-based implementations [CCK21] established the first benchmark, proving superior to Karatsuba for polynomials over 20 kbits on instruction-limited platforms. This performance lead was subsequently challenged in [CGKT22]. The authors leverage the standard 32×32→64-bit *integer* multiplication instruction for carry-less polynomial multiplication. Their optimized Karatsuba implementation outperformed previous FFT methods for polynomial degrees below 20 kbits, establishing a clear performance crossover point. The latest work [LJKH25] 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

Section 2 introduces HQC, target microarchitectures, and additive FFT. Section 3 reviews FFT-based $\mathbb{F}_2$-polymul and presents implementation techniques on targeted platforms.

Table 1: Parameter sets of HQC-KEM: Sizes of keypair $(\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{KEM}})$, ciphertext $\mathbf{c}_{\mathsf{KEM}}$ and shared key K are in Bytes.

	n	w	$w_r = w_e$	$\lvert\mathsf{ek}_{\mathsf{KEM}}\rvert$	$\lvert\mathsf{dk}_{\mathsf{KEM}}\rvert^{\dagger}$	$\lvert\mathbf{c}_{\mathsf{KEM}}\rvert$	$\lvert K\rvert$
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

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

Section 4 describes optimizing of HQC with FFT-based multiplication. Section 5 shows our implementation and performance results. Our conclusions are presented in Section 6.

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 Main components in HQC-PKE

1: Global parameters: $(n, w, w_r, w_e, \ell, \mathcal{C})$.
2: $\mathsf{HQC\text{-}PKE.Keygen}(\ \mathsf{seed}_{\mathsf{PKE}}\ )$:
3: Derive $(\mathsf{dk}_{\mathsf{PKE}}, \mathsf{seed}_{\mathsf{PKE.ek}}) \leftarrow \mathsf{HASH.I}(\mathsf{seed}_{\mathsf{PKE}})$.
4: Sample $(y, x) \xleftarrow{\mathsf{dk}_{\mathsf{PKE}}} \mathcal{R}_w \times \mathcal{R}_w$.
5: Sample $h \xleftarrow{\mathsf{seed}_{\mathsf{PKE.ek}}} \mathcal{R}$.
6: Set $\mathsf{ek}_{\mathsf{PKE}} = (\mathsf{seed}_{\mathsf{PKE.ek}}, s \leftarrow x + h \cdot y)$.
7: **return** $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}})$.
8: $\mathsf{HQC\text{-}PKE.Encrypt}(\mathsf{ek}_{\mathsf{PKE}}, m, \theta)$:
9: Sample $(r_2, e, r_1) \xleftarrow{\text{randomness } \theta} (\mathcal{R}_{w_r}, \mathcal{R}_{w_e}, \mathcal{R}_{w_r})$.
10: Sample $h \xleftarrow{\mathsf{ek}_{\mathsf{PKE}}} \mathcal{R}$.
11: Set $u \leftarrow r_1 + h \cdot r_2$.
12: Set $v \leftarrow \mathcal{C}.\mathsf{Encode}(m) + \mathrm{truncate}(s \cdot r_2 + e, \ell)$.
13: **return** $\mathbf{c}_{\mathsf{PKE}} = (u, v)$.
14: $\mathsf{HQC\text{-}PKE.Decrypt}(\mathsf{dk}_{\mathsf{PKE}}, \mathbf{c}_{\mathsf{PKE}} = (u, v))$:
15: Sample $y \xleftarrow{\mathsf{dk}_{\mathsf{PKE}}} \mathcal{R}_w$.
16: **return** $\mathcal{C}.\mathsf{Decode}(v - \mathrm{truncate}(u \cdot y, \ell))$.

Algorithm 1 depicts HQC-PKE components with main computations being multiplying in

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

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.

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 RS and RM code.) The RS code operates over the finite field $\mathbb{F}_{256}$ and performs a systematic encoding to a message. The RM code further encodes each byte of the RS codeword into a 128-bit vector, and repeats it 3 or 5 times to form the final codeword. In the decryption process, the RM code first sums the duplicated codewords and applies the maximum likelihood decoding algorithm to recover the RS codeword. The RS code then uses syndrome decoding to retrieve the message in BCH view. The code parameters are specified in [AAB+25] and the method is decribed in [AGZ].

2.2 Targeting Architectures

We evaluate our HQC implementations across both the x86 and `ARMv8-A(AArch64)` instruction set architectures (ISAs).

The Intel AVX2 instruction set provides 16 256-bit ymm registers. Since it operates as Single Instruction, Multiple Data (SIMD), the 256-bit registers can be operated as lanes of various sizes, such as 4×64, 8×32, 16×16, or 32×8-bit. We list the instructions for polynomial multiplication in x86 architecture here:

- `PCLMULQDQ` from the AESNI [Int14] instruction set performs $64 \times 64 \to 128$ multiplication in $\mathbb{F}_2[x]$. Its core application is in the Galois/Counter Mode (GCM) of AES encryption. `PCLMULQDQ` is widely supported across most modern x86 processors.
- The `GF2P8MULB` instruction, introduced in the GFNI [Int21] instruction set, performs native field multiplication in $\mathbb{F}_{256} := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$. When paired with the AVX2 instruction set, one `GF2P8MULB` instruction performs 32 field multiplications simultaneously. It was introduced in 2019 and has since been supported by newer Intel and AMD CPUs. However, its adoption is less widespread than AESNI.

Both `PCLMULQDQ` and `GF2P8MULB` can coexist in longer SIMD architectures, e.g., `VPCLMULQDQ`, requiring a different CPUID from `PCLMULQDQ`, are capable of 4 multiplications in one instruction with AVX-512 instruction set. However, we confine our HQC implementations to AVX2+`PCLMULQDQ` and AVX2+`PCLMULQDQ` +GFNI in this paper for better algorithmic comparison with official HQC optimization on AVX2+`PCLMULQDQ`.

The `ARMv8-A(AArch64)` architecture supports the NEON SIMD instruction set. ARM NEON is a 128-bit SIMD instruction set with 32 vector registers. There are two instructions dedicated to polynomial multiplications in $\mathbb{F}_2[x]$:

- The `PMULL.P8` instruction performs $8 \times 8 \to 16$ multiplication in $\mathbb{F}_2[x]$. One `PMULL.P8` processes 8 multiplications simultaneously in one 128-bit register. It is widely available even on low cost devices, e.g., Raspberry Pi4.
- The `PMULL.P64` instruction performs $64 \times 64 \to 128$ multiplication in $\mathbb{F}_2[x]$. This is the ARM equivalence instruction of `PCLMULQDQ` in the x86 world.

2.2.1 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 polymuls for the sparse polynomials in HQC to prevent leaking degrees of the terms in sparse polynomials. We also make the RM-RS decoder resistant to power side-channel attacks proposed in [PRJB23] and [MNMD25] with SIMD instructions.

2.3 Additive Fast Fourier Transform

Here is the additive FFT [Can89, GM10], which efficiently evaluates a polynomial over some additive subgroups of the coefficient field, from the viewpoint of [LCH14, LANHC16].

Binary field and Cantor basis We only handle the boolean extension fields $\mathbb{F}_{2^m}$ where m is a power of 2 in this paper. Our implementations use the fields $\mathbb{F}_{2^{64}}$, $\mathbb{F}_{256^2}$, and $\mathbb{F}_{256^{2^2}}$.

$$\mathbb{F}_{2^{64}} := \mathbb{F}_2[x]/(x^{64} + x^4 + x^3 + x + 1). \tag{2}$$

$$\mathbb{F}_{256^2} := \mathbb{F}_{256}[y]/(y^2 + y + x^5) \text{ where } \mathbb{F}_{256} := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1). \tag{3}$$

$$\mathbb{F}_{256^{2^2}} := \mathbb{F}_{256^2}[z]/(z^2 + z + x^5 y). \tag{4}$$

An element in $\mathbb{F}_{2^{64}}$ is represented as a Boolean polynomial of degree 63 or an unsigned 64-bit integer. We represent elements in the extension field $\mathbb{F}_{256^2}$ as two coefficients of a degree-1 polynomial in $\mathbb{F}_{256}[y]$ and $\mathbb{F}_{256^{2^2}}$ as degree-1 polynomials in $\mathbb{F}_{256^2}[z]$. Although we present FFT algorithms through the example of $\mathbb{F}_{2^{64}}$, the same properties apply to $\mathbb{F}_{256^2}$ and $\mathbb{F}_{256^{2^2}}$ as well.

We denote the Cantor basis in $\mathbb{F}_{2^{64}}$ as 64 particular field elements $\{v_0, \ldots, v_{63}\}$ which spans the whole linear space of $\mathbb{F}_{2^{64}}$, i.e., $\mathsf{span}\{v_0, \ldots, v_{63}\} = \mathbb{F}_{2^{64}}$, and follows the relation

$$v_i = v_{i+1}^2 - v_{i+1} \ , \tag{5}$$

and $v_0 = 1$. Let $\mathbb{F}_{2^{64}} = \mathsf{span}\{v_0, \ldots, v_{63}\}$ be an ordered set. We enumerate the set as

$$\{0, 1, v_1, v_1 + 1, v_2, \ldots, \sum_{i=0,\ldots,63} v_i\},$$

Hence, we also refer to the elements in $\mathbb{F}_{2^{64}}$ as indices of the ordered set. We call the index as a Cantor basis representation for elements in $\mathbb{F}_{2^{64}}$ in addition to a Boolean polynomial representation. Given a number i written in its binary form $i = \sum_{j=0}^{63} i_j \cdot 2^j$ where $i_j \in \{0, 1\}$, the field element referred by the index $\underline{i}$ is $\sum_j i_j \cdot v_j$. The index representation simplifies the symbols to $\mathbb{F}_{2^{64}} = \mathsf{span}\{v_0, \ldots, v_{63}\} = \{0, 1, \underline{2}, \ldots, \underline{2^{64} - 1}\}$.

Vanishing polynomials and the polynomial basis for fast evaluation We define a series of subspaces from a Cantor basis as $V_0 = \{0\}$ and

$$V_i = \mathsf{span}\{v_0, \ldots, v_{i-1}\} = \{0, \ldots, \sum_{j=0}^{i-1} v_j\} = \{0, \ldots, \underline{2^i - 1}\} \quad \text{for } i \in \{1, \ldots, 64\} \ .$$

For example, $V_1 = \mathsf{span}\{v_0\} = \{0, 1\}$, and $V_2 = \mathsf{span}\{v_0, v_1\} = \{0, 1, v_1, v_1 + v_2\} = \{0, 1, \underline{2}, \underline{3}\}$, and so on. Note that $|V_i| = 2^i$ and $V_{64} = \mathbb{F}_{2^{64}}$.

We then define the vanishing polynomial s_i of degree 2^i as the polynomial that maps all elements in the subspace V_i to 0, i.e.,

$$s_i(x) = \Pi_{j \in V_i}(x - j) = x(x - 1)(x - \underline{2}) \cdots (x - \underline{2^i - 1}) \ , \tag{6}$$

and $s_i(x) \mapsto 0$ for all $x \in V_i$. For example, $s_0(x) := \prod_{j \in V_0}(x - j) = x$, $s_1(x) := \prod_{j \in V_1}(x - j) = x(x - 1) = x^2 - x$, $s_2(x) := \prod_{j \in V_2}(x - j) = x(x - 1)(x - \underline{2})(x - \underline{3})$, etc.

We recall important properties of s_i from [GM10, LCH14].

$$\text{Linearity: } s_i(a + b) = s_i(a) + s_i(b) \ . \tag{7}$$

$$\text{Recursion: } s_i(x) = s_{i-1}(s_1(x)) \text{ for } i > 1 \ . \tag{8}$$

$$\text{Evaluation of } s_i : \ s_i(\underline{j}) = \underline{j \gg i} \ . \tag{9}$$

Note that the generating relation (5) can be seen as evaluating $s_1(x)$ at a basis element v_i:

$$s_1(v_i) = v_i^2 - v_i = v_{i-1} \quad \text{for } i \in \{1, \ldots, 63\} \ . \tag{10}$$

In [LCH14], Lin, Chung, and Han show an additive FFT algorithm for evaluating polynomials in a series of basis polynomials $\{1, X_1(x), X_2(x), X_3(x), \ldots\}$. The basis polynomial X_i of degree i is defined from vanishing polynomials s_i as

$$X_i = \prod_{j \geq 0} i_j \cdot s_j \quad \text{, where } i = \sum_{j \geq 0} i_j \cdot 2^j \text{ and } i_j \in \{0, 1\} \ . \tag{11}$$

For example, $X_0(x) := 1$, $X_1(x) := s_0(x) = x$, $X_2(x) := s_1(x) = x \cdot (x - 1)$, $X_3(x) := s_0(x) \cdot s_1(x)$, and so on.

Divide-and-conquer process for evaluating polynomials We define the additive FFT (`addFFT`) as evaluating the input polynomial on an affine subspace of Cantor basis, i.e.,

$$\texttt{addFFT}(f, a + V_i) \mapsto (f(a + 0), f(a + 1), f(a + \underline{2}), \ldots, f(a + \underline{2^i - 1})) \quad .$$

In this context, f is a polynomial in $\mathbb{F}_{2^m}[x]$ and the domain of evaluation is the affine subspace $a + V_i = \{a + u | u \in V_i\}$, where $a \in \mathbb{F}_{2^m}$ is fixed and $V_i = \texttt{span}\{1, v_1, \ldots, v_{i-1}\}$ is a linear subspace of $\mathbb{F}_{2^m}$. The function outputs the evaluated values of f over $a + V_i$.

Let f be a polynomial of degree $n - 1$ in the polynomial basis (11): $f = f_0 + f_1 X_1 + \cdots + f_{n-1} X_{n-1}$ where $n = 2^i$ is a power of 2. Suppose f is stored as a list of n coefficients $(f_0, f_1, \ldots, f_{n-1}) \in \mathbb{F}_{2^m}^n$. The `addFFT` operation outputs n evaluated values on the evaluating domain $a + V_i$ as $(f(a), f(a + 1), \ldots, f(a + \underline{n - 1}))$. The `addFFT` computation proceeds in $\lceil \log n \rceil = i$ stages, the core operation of each being a recursive step reducing the problem size.

In each stage, f is partitioned into two sub-polynomials as $f = f_l + X_{n/2} \cdot f_h$ by splitting the coefficient list into $(f_0, \ldots, f_{n/2-1})$ and $(f_{n/2}, \ldots, f_{n-1})$. Since n is a power of 2, $X_{n/2} = s_{i-1}$. The linear space V_i is similarly decomposed into two disjoint sets: V_{i-1} and $v_{i-1} + V_{i-1}$. By utilizing the properties: $s_{i-1}(v_{i-1}) = 1$ and $s_{i-1}(V_{i-1}) = 0$, the evaluation of f on $a + V_i$ is reduced to two smaller evaluations: $\texttt{addFFT}(f_l + s_{i-1}(a) f_h, a + V_{i-1}))$ and $\texttt{addFFT}(f_l + (s_{i-1}(a) + 1) f_h, a + v_{i-1} + V_{i-1})$. Consequently, the output of $\texttt{addFFT}(f, a + V_i)$ is the concatenation of the results from these two recursive calls.

A so-called 'Butterfly' operation computes terms $f_l + s_{i-1}(a) \cdot f_h$ and $f_l + (s_{i-1}(a) + 1) \cdot f_h$ in each stage. Figure 1 visualizes a butterfly operation. Only one multiplication by $s_{i-1}(a)$ is performed in each butterfly unit.

Algorithm 2 presents the `addFFT` procedure. It takes two inputs, which are a polynomial f and the evaluation domain $a + V_i$, and performs the following operations:

1. `BasisCvt`: Converting the polynomial basis of f from monomial basis to the polynomial basis (11). This step has $O(n \log n \log \log n)$ complexity [LANHC16] but can be skipped if the polynomial f is already in the polynomial basis (11).

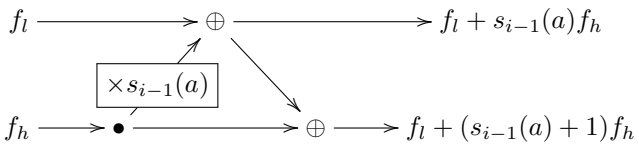

Figure 1: The butterfly operation

Algorithm 2 The `addFFT` algorithm

```
addFFT(f, a + V_i ):                          ▷ f is a polynomial of degree n − 1 and n = 2^i.
    if f in monomial basis then
        f ← BasisCvt(f).                      ▷ Converting to polynomial basis (11)
    return Butterfly(f, a + V_i).
Butterfly(f, a + V_i):
    Let f = f_l + s_{i−1} · f_h.              ▷ f_l and f_h are polynomials of degrees n/2 − 1.
    if i = 1 then
        return (f_l + a · f_h, f_l + (a + 1) · f_h).
    Set f'_l ← f_l + s_{i−1}(a) · f_h.
    Set f'_h ← f_l + (s_{i−1}(a) + 1) · f_h.
    return (Butterfly(f'_l, a + V_{i−1}), Butterfly(f'_h, a + v_{i−1} + V_{i−1})).
```

2. `Butterfly`: Perform $\log n$ stages of butterfly operations (Figure 1) on the converted polynomial f for evaluating on $c + V_i$. Since each stage computes $n/2$ butterflies operation, this step has $O(n \log n)$ complexity.

The presented `addFFT` has complexity $O(n \log n)$ for a polynomial in the basis (11). However, if the polynomial is in the monomial basis, the fastest `addFFT` requires $O(n \log n \log \log n)$ complexity because converting the polynomial basis dominates its asymptotic complexity.

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

We categorize polynomial multiplication algorithms in $\mathbb{F}_2[x]$ by polynomial length. The first category, base multiplication, directly performs multiplication operations using hardware instructions. This is typically employed when the length of the polynomials being multiplied is equal to the machine word size of the underlying architecture.

The second category of algorithms is more suitable for medium-sized polynomials, particularly when base multiplication method (tied to the underlying architecture's word size) has already been determined. For instance, the official HQC implementation utilizes a hybrid Toom-Karatsuba algorithm (see [Knu97]) for multiplying polynomials of specific lengths. The Karatsuba algorithm divides a multiplication of $(A_0 + A_1 x^n) \cdot (B_0 + B_1 x^n)$ into three multiplications of $A_0 \cdot B_0$, $A_1 \cdot B_1$, and $(A_0 + A_1) \cdot (B_0 + B_1)$ by $(A_0 + A_1 x^n) \cdot (B_0 + B_1 x^n) = A_0 B_0 + [(A_0 + A_1) \cdot (B_0 + B1) + A_0 B_0 + A_1 B_1] x^n + A_1 B_1 x^{2n}$. The algorithm is applied recursively until the length of the divided polynomials is equal to the base multiplication. Its asymptotic complexity is $O(n^{\log_2 3})$. The Toom-Cook algorithm evaluates divided parts of input polynomials on some easily computable points and interpolates the products of evaluated values into its polynomial form. Toom and Karatsuba are usually considered simpler with a higher asymptotic complexity compared to the last category.

The last category is FFT-based algorithms, which provide the lowest asymptotic complexity and are considered good for polynomials of "large" degrees. We present these FFT-based algorithms for polynomial multiplication in Subsection 3.1.

The distinction between "medium" and "large" polynomial degrees, which dictates the choice of algorithm, is not fixed. The practical crossover point, where FFT-based methods become more efficient than Toom-Karatsuba like algorithms, is influenced by the instruction set available on the underlying hardware architecture. For example, a processor equipped with wide carry-less multiplication instructions (e.g., 64x64-bit `PCLMULQDQ`) will favor Karatsuba for much longer polynomials compared to a processor limited to shorter multiplication instructions (e.g., 8x8-bit `PMULL.P8`). Therefore, the ideal algorithm is a function of both the polynomial degree and the specific hardware capabilities.

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

A primary performance bottleneck in HQC is the multiplication within its defined polynomial ring, $\mathcal{R} := \mathbb{F}_2[x]/(x^n + 1)$. More specifically, a key challenge is that the parameter n is not a power of two in the settings of HQC, which makes standard FFT-based polymuls inefficient. To overcome this obstacle, our strategy is twofold. First, we employ highly efficient algorithms based on `addFFT` to handle the large, power-of-two portion of the polynomial multiplication. Second, to address the non-power-of-two degrees, we introduce a split of the polynomial ring for the combination of additive FFT with the Chinese Remainder Theorem (CRT). This approach allows us to decompose one large, irregular multiplication problem into two smaller, more regular sub-problems. In doing so, we can leverage the power of `addFFT` where it is most effective, while bypassing the limitations imposed by the polynomial's specific degree.

This section is organized as follows: Subsection 3.1 describes two practical methods for using `addFFT` to multiply polynomials in $\mathbb{F}_2[x]$: Kronecker Segmentation (KS) and the Frobenius Additive FFT (FAFFT). Subsection 3.2 details our new technique for combining `addFFT` with the CRT to efficiently handle the non-power-of-two polynomial degrees in HQC. Finally, Subsection 3.3 synthesizes these techniques, compares their estimated computational cost against the previous Toom-Karatsuba method. The performance benchmarks with regard to various platforms are presented in Section 5.

3.1 FFT-based Polynomial Multiplication in $\mathbb{F}_2[x]$

We introduce two FFT-based methods for multiplying polynomials in $\mathbb{F}_2[x]$ in this section.

3.1.1 Kronecker Segmentation

The basic FFT-based method for multiplying Boolean polynomials is Kronecker segmentation (KS). Because the polynomial degree plays an important role on describing the method, we denote the polynomials of degree less than n as $\mathbb{F}_2[x]_{<n}$. Given two polynomials $A, B \in \mathbb{F}_2[x]_{<n}$, we compute $C = A \cdot B \in \mathbb{F}_2[x]_{<2n}$ using KS as follows:

1. Partition A and B into m-bit blocks, i.e., $A = \sum_0^{n-1} a_i x^i = \sum_0^{n/m-1} \bar{a}_i y^i$ where $\bar{a}_i \in \mathbb{F}_2[x]_{<m}$ and $y = x^m$.
2. Lift coefficients $\bar{a}_i \in \mathbb{F}_2[x]_{<m}$ into $\mathbb{F}_{2^{2m}}$ and let $\bar{A} = \sum_0^{n/m-1} \bar{a}_i y^i \in \mathbb{F}_{2^{2m}}[y]_{<n/m}$.
3. Compute the product $\bar{C} \in \mathbb{F}_{2^{2m}}[y]_{<2n/m} = \bar{A} \cdot \bar{B}$ by using a standard FFT-based polynomial multiplication over $\mathbb{F}_{2^{2m}}[y]$ as follows:
   (a) Forward FFT transform: $\hat{A} = (a'_0, \ldots, a'_{2n/m-1}) \in \mathbb{F}_{2^{2m}}^{2n/m} \leftarrow \texttt{addFFT}(\bar{A}, V_{\log(2n/m)})$.
   (b) Forward FFT transform: $\hat{B} = (b'_0, \ldots, b'_{2n/m-1}) \in \mathbb{F}_{2^{2m}}^{2n/m} \leftarrow \texttt{addFFT}(\bar{B}, V_{\log(2n/m)})$.

(c) Pointwise multiplication on two evaluated sets:

$$\hat{C} \in \mathbb{F}_{2^{2^m}}^{2n/m} \leftarrow \hat{A} \otimes \hat{B} = (a'_0 \cdot b'_0, \ldots, a'_{2n/m-1} \cdot b'_{2n/m-1}) \ .$$

(d) Inverse FFT transform: $\bar{C} \in \mathbb{F}_{2^{2^m}}[y]_{<2n/m} \leftarrow \texttt{addFFT}^{-1}(\hat{C}, V_{\log(2n/m)})$.

4. Map $\bar{C} \to C \in \mathbb{F}_2[x]_{<2n}$ by substituting $y = x^m$.

Note that we split the polynomials into blocks of size m and compute in $\mathbb{F}_{2^{2^m}}$ to prevent "overflow". This also increases the required storage space for storing coefficients.

In our implementation for HQC, we apply the KS method to multiply polynomials with degrees ranging from 2^{14} to 2^{16} bits. The polynomials are partitioned into 8-bit blocks, which means the operations are performed in a field of size 2^{16}. However, instead of a standard $\mathbb{F}_{2^{16}}$, representation, we follow the strategy from [CCK+17] for using an extension field, specifically $\mathbb{F}_{256^2}$. This choice offers two main advantages:

- Efficient Arithmetic: The field $\mathbb{F}_{256}$ is based on an 8-bit polynomial representation. This specific size is highly suitable for leveraging short polynomial multiplication instructions, e.g., `PMULL.P8`.
- Faster Butterfly Operations: While computing `addFFT`, many of the constant multipliers are elements of the subfield $\mathbb{F}_{256}$. For example, for any $a \in \mathbb{F}_{2^{16}}$, $s_i(a) \in \mathbb{F}_{2^8}$ for stages of $i \geq 8$ due to Eq. (9). Multiplying elements of $\mathbb{F}_{256^2}$ by an element from this subfield only requires two multiplications in $\mathbb{F}_{256}$, making these initial FFT stages faster than if they were performed in a pure $\mathbb{F}_{256^2}$ field.

The main drawback of this approach is the need to perform a field isomorphism to map elements from $\mathbb{F}_{2^{16}}$ to $\mathbb{F}_{256^2}$, which is implemented as a 16×16 linear transformation.

3.1.2 Frobenius Additive FFT

A naive FFT polynomial multiplication With the `addFFT` algorithm, we can naively perform multiplications in $\mathbb{F}_2[x]$: given polynomials $A, B \in \mathbb{F}_2[x]_{<n}$ such that $C = A \cdot B$ is of degree smaller than $2n = 2^{l+1}$, we can lift A and B to $\mathbb{F}_{2^m}[x]_{<n}$ such that m is a power of 2 and $l + 1 \leq m$. We perform $\hat{A} = \texttt{addFFT}(A, V_{l+1})$ and $\hat{B} = \texttt{addFFT}(B, V_{l+1})$, pointwise multiplication, and inverse `addFFT` to obtain $C = \texttt{addFFT}^{-1}(\hat{A} \otimes \hat{B}, V_{l+1})$. This FFT-based multiplication works. However, it is less efficient than the KS method because it maps only one $\mathbb{F}_2$ coefficient to one $\mathbb{F}_{2^m}$. The KS method packs $m/2$ coefficients from $\mathbb{F}_2$ into a single $\mathbb{F}_{2^m}$ coefficient.

In 2017, Hoeven and Larrieu [vdHL17] improved the naive (multiplicative) FFT-based multiplication with the help of Frobenius map ϕ_2 (squaring in $\mathbb{F}_2$): Given $a \in \mathbb{F}_{2^m}$, the value $A(\phi_2(a))$ can be deduced from $\phi_2(A(a))$, meaning one can evaluate A on a smaller set and then deduce values outside the set with ϕ_2.

Frobenius additive FFT In prior work [LCK+18, CCK+18], the authors proposed the evaluation domain for `addFFT` to be

$$\Sigma = v_{l'+m/2} + V_{l'} \tag{12}$$

subject to the conditions

$$l' + m/2 < m \ , \quad n' = |\Sigma| = \frac{2n}{m} \ , \text{and} \quad l' = \log_2 n' < \frac{m}{2} \ .$$

The key finding is that the evaluation map $\texttt{addFFT}(\cdot, \Sigma)$, which transforms a polynomial f into $f(\Sigma)$, is invertible. This is because the specific design of Σ includes a "gap" of $m/2$

between the element $v_{l'+m/2}$ and the space $V_{l'}$. This gap enables the $2n/m$ evaluation values (obtained from $\texttt{addFFT}(\cdot, \Sigma)$) to be converted into $2n$ elements in the field $\mathbb{F}_{2^m}$ using the Frobenius map. It means that we can accomplish the multiplications in $\mathbb{F}_2[x]$ by evaluating A, B only on the set Σ and interpolating their product C from value space back to coefficients successfully. This way, the number of evaluation points is reduced by a factor of m compared to the naive FFT multiplication. They called the method of performing addFFT on the set Σ as Frobenius additive FFT (FAFFT).

While interpolating from the $2n/m$ evaluations in Σ to $2n$ coefficients in $\mathbb{F}_{2^m}$, we actually perform the inverse $\texttt{addFFT}$ on the larger domain $\tilde{\Sigma} = v_{l'+m/2} + V_{l'+\log_2 m}$ such that $\Sigma \subset \tilde{\Sigma}$ and $|\tilde{\Sigma}| = 2n$. Hence, we present the full polynomial multiplication with FAFFT as :

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. Lift the coefficient ring of the polynomials $A, B \in \mathbb{F}_2[x]_{<n}$ to $A, B \in \mathbb{F}_{2^m}[x]_{<n}$.
2. Compute additive FFT transforms: $\hat{A}, \hat{B} \in \mathbb{F}_{2^m}^{2n/m} = \texttt{addFFT}(A, \Sigma), \texttt{addFFT}(B, \Sigma)$.
3. Pointwise multiplication: $\hat{C} \in \mathbb{F}_{2^m}^{2n/m} \leftarrow \hat{A} \otimes \hat{B}$.
4. Inverse FFT transform: $C \in \mathbb{F}_{2^m}[x]_{<2n} \leftarrow \texttt{addFFT}^{-1}(\hat{C}, \tilde{\Sigma})$.
5. Map the coefficients of C to $\mathbb{F}_2$ such that $C \in \mathbb{F}_2[x]_{<2n}$.

The complexity of the polynomial multiplication is the same as the FFT in use.

Encode: a truncated computation of addFFT When performing the computation $\texttt{addFFT}(A, \Sigma)$ of FAFFT, there is a mismatch between the number of coefficients of A and the number of evaluating points $|\Sigma| = 2n/m$. FAFFT see this as an evaluation on a matched set $\texttt{addFFT}(A, \tilde{\Sigma})$ with truncation of the unnecessary values of points outside Σ. The starting $\log m$ butterfly stages of this truncated additive FFT, which reduces the starting $2n$ points to $2n/m$, are seen as an **Encoding** operation:
With respect to $\tilde{\Sigma}$, **Encoding** is an invertible linear map which maps coefficients of $f(x)$, e.g., $(f_0, f_1, \ldots, f_{2n-1}) \in \mathbb{F}_2^{2n}$ to $(f'_0, \ldots, f'_{n'-1}) \in \mathbb{F}_{2^m}^{n'}$ where the f'_i is defined as

$$f'_i = \sum_{j=0}^{m-1} r_j \cdot f_{j \cdot n' + i}, \quad r_j = \prod_{k=0}^{\log m - 1} (v_{m/2-k})^{j_k}$$

where $(j_0, j_1, \ldots, j_{\log m - 1}) \in \{0,1\}^{\log m}$ is the binary representation of the integer j, i.e., $j = \sum_{k=0}^{\log m - 1} j_k \cdot 2^k$. The constants r_j are derived from the accumulation of constant multipliers across stages of the $\texttt{addFFT}$ algorithm. Specifically, r_j is defined as the product of the constant multipliers $s_{i-1}(a)$ from $\log m$ stages of the $\texttt{Butterfly}$ operation, detailed in Algorithm 2. The initial constant a for FAFFT starts with $a = v_{l'+m/2}$ as defined in the domain Σ (Eq. (12)).

3.1.3 Comparison of FFT-based Multiplication Methods

Both the FAFFT and KS methods share the same subroutines and asymptotic complexity $O(n \log n \log\log n)$, reflecting the complexity of $\texttt{addFFT}$ operations. However, they differ in practical efficiency and memory usage.

The primary advantage of FAFFT lies in its computational and memory efficiency. When multiplying polynomials to produce a product of $2n$ bits, FAFFT performs its transform on an evaluation domain of size $2n/m$ over the field $\mathbb{F}_{2^m}$. In contrast, the KS method

partitions the $2n$ bits into m-bit blocks and operates on a domain of the same size $(2n/m)$ but over the larger field $\mathbb{F}_{2^{2m}}$. Consequently, FAFFT requires approximately half the storage space for the transformed polynomial values and subsequently about half the run time for the transform and pointwise multiplication steps.

However, while multiplying polynomials of the same size, the KS method can usually operate with smaller fields. For instance, multiplying polynomials with a degree approaching 2^{16} bits is not feasible with FAFFT over $\mathbb{F}_{2^{16}}$ because the required domain size $|\tilde{\Sigma}| = 2^{17}$ would exceed the field size 2^{16}. In contrast, KS can partition the 2^{16} bits into 2^{13} byte segments and perform an `addFFT` for a polynomial in $\mathbb{F}_{2^{16}}[x]$ over a domain V_{14}.

In summary, FAFFT is generally faster and more memory-efficient as long as the polynomial degree is well within the limits imposed by the chosen field size. KS, while requiring more storage and operations, allows for operation in a smaller field that can be chosen to align with the underlying instruction set.

3.2 Combination of Additive FFT and Chinese Remainder Theorem

The FFT-based multiplications are efficient for polynomials of degrees that are powers of 2. However, the polynomial lengths, or degrees, used in HQC are not precisely powers of 2. This discrepancy poses a challenge, as direct application of standard FFT algorithms becomes inefficient. To overcome this, a new technique combining additive FFT and Chinese Remainder Theorem (CRT) is employed. This approach allows for the efficient multiplication of polynomials with degrees of the form $2^k + c$, where 2^k 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:

$$\mathbb{F}_2[x]_{<2^k+c} \xleftrightarrow{\text{CRT}} \mathbb{F}_2[x]/s_k \times \mathbb{F}_2[x]/x^c \ .$$

This decomposition enables the use of efficient additive FFT for the ring $\mathbb{F}_2[x]/s_k$ and a simpler modular reduction for the $\mathbb{F}_2[x]/x^c$ ring, thereby optimizing the overall multiplication process for non-power-of-2 degrees. While a similar CRT decomposition utilizing $\mathbb{F}[x]/x^c$ as one of the smaller rings was previously proposed for polynomial multiplication over prime fields by [CYW25], our approach is fundamentally distinct. We are the first to adapt this decomposition to binary fields and combine it with the additive FFT framework instead of NTT in this work.

While CRT usually requires moduli to be coprime, a generalized version can be applied even when moduli share a common factor, provided certain conditions are met. This section clarifies some key properties relevant to our application of CRT with additive FFT.

Proposition 1. $\texttt{GCD}(s_i, x^c) = x$.

Since $s_i = x \cdot \Pi_j(x - j)$ for all $j \in V_i$ and $j \neq 0$ by (6) and x^c by definition has x as its only irreducible factor, their Greatest Common Divisor (GCD) is x. .

Proposition 2. CRT works despite $\texttt{GCD}(s_i, x^c) = x \neq 1$.

A challenge with this CRT-based decomposition is that the moduli, s_i and x are not coprime. As in Proposition 1, they share a common factor of x. Therefore, the standard Chinese Remainder Theorem does not directly apply.

However, a generalized version of the CRT can be used. For a polynomial f to be uniquely determined by its remainders, $c_0 = f \mod x^c$ and $c_1 = f \mod s_i$, a consistency condition must be met: the two remainders must be congruent modulo their GCD, x, i.e.,

$$c_0 \equiv c_1 \pmod{x} \ ,$$

requiring c_0 and c_1 have the same constant term.

This consistency condition is satisfied in our polymul application, where we compute $f = g \cdot h$. The constant terms of the remainders are guaranteed to be identical as:

- The modulus s_i is a multiple of x by its definition (6), so the operation $f \mod s_i$ does not change the constant term of f.
- Similarly, the operation $f \mod x^c$ does not change the constant term.

Since both remainders c_0 and c_1 preserve the original constant term of the product f, they have the same constant term.

Proposition 3. $\texttt{addFFT}(f, V_i) = \texttt{addFFT}(f \mod s_i, V_i)$.

The vanishing polynomial s_i is defined such that its roots are precisely the set of evaluation points V_i, i.e., for any $v \in V_i$, $s_i(v) = 0$. If we have a polynomial f and we divide f by s_i to obtain $f = Q \cdot s_i + R$, where $R = f \mod s_i$ is the remainder with $\deg R < \deg s_i$. For any evaluation point $v \in V_i$, we can substitute v into the equation:

$$f(v) = Q(v)s_i(v) + R(v) \ .$$

Since $s_i(v) = 0$ for all $v \in V_i$, the equation simplifies to :

$$f(v) = Q(v) \cdot 0 + R(v) = R(v) \ .$$

This demonstrates that f and its remainder $f \mod s_i$ evaluate to the same values at all points in V_i. This proposition implies we cannot apply the same setting to FAFFT since FAFFT evaluates on points outside V_i when used for polynomial multiplication.

The reverse CRT map The Reverse CRT map is required to combine the results obtained from the two smaller modular rings, $(\mathbb{F}_2[x]/s_i \times \mathbb{F}_2[x]/x^c)$, in order to recover the desired polynomial f in the larger ring $\mathbb{F}_2[x]_{<2^i+c}$.

Specifically, given the results c_0 and c_1 from the smaller rings (where $\deg c_0 < c$ and $\deg c_1 < 2^i$), we seek a polynomial $f \in \mathbb{F}_2[x]_{<c+2^i-1}$ satisfying:

$$\begin{aligned} f &\equiv c_0 \pmod{x^c} \\ f &\equiv c_1 \pmod{s_i} \ . \end{aligned}$$

To find the polynomial f, the reverse CRT map is executed in two steps:

1. We first determine the auxiliary polynomial $k \in \mathbb{F}_2[x]_{<c-1}$ via this congruence:

$$k \equiv I_s \cdot (c_0 - c_1)/x \pmod{x^{c-1}} \ . \tag{13}$$

   This step involves multiplying a pre-computed polynomial I_s (the inverse of s_i/x modulo x^{c-1}) Note that only the first $c-1$ terms of $(c_0 - c_1)/x$ are required in the computation since $\deg I_s < c - 1$.

2. Once k is determined, the final result f is recovered directly using the definition:

$$f = c_1 + k \cdot s_i \ . \tag{14}$$

   The polynomial f lies in the target ring $\mathbb{F}_2[x]_{<c+2^i-1}$.

The derivation of Equation (13) is based on applying the extended Euclidean algorithm to Proposition 1, which establishes the relation

$$1 \equiv I_s \cdot (s_i/x) \pmod{x^{c-1}} \ .$$

By substituting $f = c_1 + k \cdot s_i$ into the first congruence: $f \equiv c_0 \pmod{x^c}$, we arrive at

$$k \cdot (s_i/x) \equiv (c_0 - c_1)/x \pmod{x^{c-1}} \ .$$

Since $c_0 - c_1$ is divisible by x by Proposition 2, dividing by x and multiplying by the inverse I_s yields the expression for k in Equation (13).

For optimization, we employ a product scanning technique to compute the resulting terms of f. This optimization collects all required source terms from two input polynomials (c_0 and c_1) for a specific output degree. This approach significantly reduces redundant memory accesses to the destination terms, which is a bottleneck during the CRT computation.

3.3 Polynomial Multiplication in HQC

In this section, we detail the steps for combining the KS and CRT techniques for polynomial multiplication in HQC. We use HQC-1 as a concrete example. In HQC-1, the polynomial length is $n = 17669$. The steps are as follows:

1. **Lift to $\mathbb{F}_2[x]_{<32768+3072}$**: The product of two polynomials has a degree less than $2n = 35338$. To apply our method, we choose the nearest power of two, $2^{15} = 32768$, and handle the remaining portion. Our multiplication operates on polynomials in $\mathbb{F}_2[x]_{<32768+3072}$ to produce a product of degree up to $32768 + 3072 = 35840$.

2. **Map to $\mathbb{F}_{2^{16}}[x]_{<4096+384}$ by Kronecker segmentation** : The lifted polynomial in $\mathbb{F}_2[x]$ is then segmented into one-byte blocks, and the one-byte data are mapped into coefficients of polynomials in the ring $\mathbb{F}_{2^{16}}[x]$. Since all mapped coefficients are single-byte, this step in effectively a null operation.

3. **Map to $\mathbb{F}_{2^{16}}[x]/s_{12} \times \mathbb{F}_{2^{16}}[x]/x^{384}$ by CRT** : Using the CRT, the polynomial in $\mathbb{F}_{2^{16}}[x]_{4096+384}$ is decomposed into two components. The first component is modulo $s_{12} = x^{4096} + \ldots$. Since $\deg s_{12} = 4096 > (17669/8)$, degree of input polynomial in $\mathbb{F}_{2^{16}}$, this map is conceptual and has nothing to do again. The second component is modulo x^{384}, which involves a simple truncation and can be multiplied directly.

4. **Multiplication in the two rings: $\mathbb{F}_{2^{16}}[x]/s_{12}$ and $\mathbb{F}_{2^{16}}[x]/x^{384}$**: The multiplication on $\mathbb{F}_{2^{16}}[x]/x^{384}$ is done with a *truncated* Karatsuba algorithm, truncating coefficients of degree ≥ 384. Since the coefficients are only one byte, the multiplication between coefficients will not induce the "reduce" operation in the field multiplication, which is more efficient than normal field multiplication.

   We use KS for the multiplication in $\mathbb{F}_{2^{16}}[x]/s_{12}$. Again, we extract the one-byte coefficients to perform `BasisCvt` in the byte-data format. After `BasisCvt` and before `Butterfly`, we change this field representation of input polynomials from $\mathbb{F}_{2^{16}}[x]$ to $\mathbb{F}_{256^2}[x]$ (field isomorphism). This way, given an input polynomial $g \in \mathbb{F}_{256^2}[x]$, we perform $\texttt{addFFT}(g, V_{12})$ and exploit the fast subfield multiplication described in 3.1.1. After the pointwise multiplication, inverse `addFFT` are finished, we change the field representation of the product back to $\mathbb{F}_{2^{16}}[x]$.

5. **Map the two products back to ring on $\mathbb{F}_{2^{16}}[x]_{<4096+384}$ by CRT** :

   (a) Find $k = I_s \cdot (c0 - c1)/x \pmod{x^{383}}$ by (13). We need only 383 terms of $(c0 - c1)/x$ and multiply I_s with degree < 383 for k

   (b) Find $f = c_1 + k \cdot s_{12}$ by (14).

6. **Recover the product to $\mathbb{F}_2[x]_{<32768+3072}$.**

On the contrary, if we use FAFFT, we partition the input polynomials in $\mathbb{F}_2[x]_{<16384+1536}$ into 2 parts. The first part comprises terms of degrees < 16384, and we perform their

Table 2: Comparison of estimating numbers of base multiplications

Implementation	Base multiplication	n =17669	n =35851	n =57637
[AAB+25]	`PCLMULQDQ`	7290	21 870	54 675
Proposed FFT-based methods				
Algorithm	Base multiplication	16 384 + 1536	32 768 + 4096	65 536
FAFFT	$\mathbb{F}_{2^{64}}$ + `PCLMULQDQ`	7424 + 3672	16 384 + 12 393	35 840 + 0
FAFFT	$\mathbb{F}_{256^{2^2}}$ + `PCLMULQDQ`	16 384 + 3672	35 840 + 12 393	77 824 + 0
KS+CRT	$\mathbb{F}_{256^2}$ + `PMULL.P8`	77 824 + 3240	167 936 + 17 496	360 448 + 0

multiplication by FAFFT. The second part comprises terms of degrees ≥ 16384, which are relatively short polynomials. Their multiplications are performed with Karatsuba. However, we have to perform the multiplication for all the cross terms between the two parts, which can be avoided by using the CRT method.

We process HQC-3 which $n = 35851 < 32768 + 4096$ with similar methods as HQC-1. Now the ring after KS becomes $\mathbb{F}_{2^{16}}[x]_{<8192+1024}$ and the two rings for CRT are $\mathbb{F}_{2^{16}}[x]/(s13) \times \mathbb{F}_{2^{16}}[x]/(x^{1024})$. The multiplication in the ring $\mathbb{F}_{2^{16}}[x]/(s13)$ is performed by addFFT multiplication over $\mathbb{F}_{256^2}[x]$.

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 FFT-based multiplication in the previous sections.

Summary Table 2 summarizes our proposed FFT-based implementations, detailing the estimated number of base multiplications required for multiplying Boolean polynomials of different degrees. These estimates are derived by summing the operations across three major components: the addFFT transforms, the pointwise multiplications, and the Karatsuba combination steps applied to the remaining segments. We note that the CRT operation consumes only addition costs and involves no multiplication overhead.

In Toom-Karatsuba, the numbers are measured from the implementation in [AAB+25]. In the implementation, the base multiplication is 64×64→128 bits polynomial multiplication, which are `PCLMULQDQ` and `PMULL.P64` in x86 and ARMv8 architectures, respectively.

In our FFT-based implementations, since the `addFFT` algorithm is suitable for processing polynomials of 2-power-sized, the 17669, 35851, and 57 637 bits of polynomials are processed as 16384+1536, 32768+4096, and 65 536 bits polynomials respectively. The long segments of 2-powered bits are multiplied with FFT-based algorithms, either FAFFT or KS. The remaining short segments are processed with Karatsuba.

In FAFFT implementation, the base multiplication is field multiplication in $\mathbb{F}_{2^{64}}$ or $\mathbb{F}_{256^{2^2}}$ for the `addFFT` computation and `PCLMULQDQ` for the remaining Karatsuba segment. In KS+CRT implementation, the base multiplication is field multiplication in $\mathbb{F}_{256^2}$ for `addFFT` computation and `PMULL.P8` for the multiplication in the ring $\mathbb{F}_{2^{16}}[x]/x^c$.

4 Optimizing HQC

While FFT-based multiplication may not always significantly outperform a highly-optimized Karatsuba algorithm in raw performance, we can leverage its unique computational structure to achieve substantial overall performance gains in HQC's encapsulation and decapsulation operations. The optimization methods we propose, as detailed in Algorithm 3, focus on two key strategies: cached key transform and common input transforms. We

note that the two strategies are similar to those used in NTT-based schemes such as ML-KEM/Kyber [SAB+22], which also cache values in the transformed domain.

Algorithm 3 HQC-PKE with `addFFT`

Note: (1) We use the term ***addFFT*** *to denote the input transform operation shared by both the FAFFT and KS methods. We omit the specific evaluation domains of these two methods for brevity. (2) Blue text indicates a reused input transform.*

Global parameters: $(n, w, w_r, w_e, \ell, \mathcal{C})$.
HQC-PKE.KeygenCache($\mathsf{seed}_{\mathsf{PKE}}$):
  Derive $(\mathsf{dk}_{\mathsf{PKE}}, \mathsf{seed}_{\mathsf{PKE.ek}}) \leftarrow \mathsf{HASH.I}(\mathsf{seed}_{\mathsf{PKE}})$.
  Sample $(y, x) \xleftarrow{\mathsf{dk}_{\mathsf{PKE}}} \mathcal{R}_w \times \mathcal{R}_w$.
  Sample $h \xleftarrow{\mathsf{seed}_{\mathsf{PKE.ek}}} \mathcal{R}$.
  Set $\hat{h}, \hat{y} = \texttt{addFFT}(h), \texttt{addFFT}(y)$.
  Set $\hat{s} = \texttt{addFFT}(x + \texttt{addFFT}^{-1}(\hat{h} \otimes \hat{y}))$.
  Set $\mathsf{ek}_{\mathsf{CPKE}}, \mathsf{dk}_{\mathsf{CPKE}} = (\mathsf{seed}_{\mathsf{PKE.ek}}, \hat{s}, \hat{h}), (\mathsf{dk}_{\mathsf{PKE}}, \hat{y})$.
  return $(\mathsf{ek}_{\mathsf{CPKE}}, \mathsf{dk}_{\mathsf{CPKE}})$.
HQC-PKE.EncryptCache($\mathsf{ek}_{\mathsf{CPKE}}, m, \theta$):
  Sample $(r_2, e, r_1) \xleftarrow{\text{randomness } \theta} (\mathcal{R}_{w_r}, \mathcal{R}_{w_e}, \mathcal{R}_{w_r})$.
  Set $\hat{r_2} = \texttt{addFFT}(r_2)$.
  Set $u \leftarrow r_1 + \texttt{addFFT}^{-1}(\hat{h} \otimes \hat{r_2})$.
  Set $v \leftarrow \mathcal{C}.\mathsf{Encode}(m) + \mathrm{truncate}(\texttt{addFFT}^{-1}(\hat{s} \otimes \hat{r_2}) + e, \ell)$.
  return $\mathbf{c}_{\mathsf{PKE}} = (u, v)$.
HQC-PKE.DecryptCache($\mathsf{dk}_{\mathsf{CPKE}}, \mathbf{c}_{\mathsf{PKE}} = (u, v)$):
  return $\mathcal{C}.\mathsf{Decode}(v - \mathrm{truncate}(\texttt{addFFT}^{-1}(\texttt{addFFT}(u) \otimes \hat{y}), \ell))$.

4.1 Cached Key Transforms

A primary overhead in FFT-based multiplication comes from the "input transform" process, which converts a polynomial from its coefficient representation to evaluation value representation. To minimize this overhead, we introduce a key caching strategy.

As shown in HQC-PKE.KeygenCache in Algorithm 3, we can perform a one-time preprocessing step on the keys, either during key generation or before their **first use**, while still preserving the original key format. Specifically:

- The polynomials h and s in the encryption key are transformed and stored in their value representations, $\hat{h}$ and $\hat{s}$, in the cached encryption key $\mathsf{ek}_{\mathsf{CPKE}}$. The $\mathsf{ek}_{\mathsf{CPKE}}$ format simply appends two auxiliary terms ($\hat{h}$ and $\hat{s}$) to the original $\mathsf{ek}_{\mathsf{PKE}}$.
- Similarly, the decryption key polynomial y is transformed to $\hat{y}$ and appended to the original $\mathsf{dk}_{\mathsf{PKE}}$ as $\mathsf{dk}_{\mathsf{CPKE}}$.

These cached value representations – $(\hat{h}, \hat{s})$ in $\mathsf{ek}_{\mathsf{CPKE}}$ and $\hat{y}$ in $\mathsf{dk}_{\mathsf{CPKE}}$ – can then be used directly in subsequent encryption and decryption operations. The main advantage is that the costly `addFFT` input transforms are not performed repeatedly during frequent operations, thus amortizing their computational cost. The other benefit is that we remove the sampling process for polynomials h and y in HQC-PKE.Encrypt and HQC-PKE.Decrypt, respectively.

We note that the cached key format requires more storage space compared to the original keys (see Table 3). This storage increase is attributed to the transformed value representations. Specifically, the polynomial s (of size $|s|$) is transformed into its cached form $\hat{s}$

(of size $|\hat{s}|$). The cached polynomial size $|\hat{s}|$ is approximately $2\times$ the original size $|s|$ for FAFFT, and approximately $4\times$ for KS.

However, in many application scenarios – such as a server handling numerous requests with the same key pair – this time-for-space tradeoff is highly effective. We emphasize that cached keys are solely designed to accelerate encapsulation and decapsulation operations. Since they are pre-processed locally from the original keys, they are not intended for communication, meaning the increased size only affects local storage. Crucially, as the cached data contains the original key material, it requires the same level of cryptographic protection as the original keys.

Table 3: Sizes (in bytes) of polynomials and cached keys

	Standard			FAFFT			KS+CRT		
	$\|s\|$	$\|\mathsf{ek}_{\mathsf{KEM}}\|$	$\|\mathsf{dk}_{\mathsf{KEM}}\|$	$\|\hat{s}\|$	$\|\mathsf{ek}_{\mathsf{CKEM}}\|$	$\|\mathsf{dk}_{\mathsf{CKEM}}\|$	$\|\hat{s}\|$	$\|\mathsf{ek}_{\mathsf{CKEM}}\|$	$\|\mathsf{dk}_{\mathsf{CKEM}}\|$
HQC-1	2209	2241	2321	4096+2240	12 673	19 089	8192+384	19 009	27 665
HQC-3	4482	4514	4602	8192+4608	25 506	38 394	16384+1024	38 306	55 802
HQC-5	7205	7237	7333	16 384	40 005	56 485	32 768	72 773	105 637

4.2 Encryption Optimization: Common Input Transform

Our second optimization strategy takes full advantage of the internal structure of HQC-PKE. In HQC-PKE.Encrypt from Algorithm 1, the random polynomial r_2 is used to compute two separate parts of the ciphertext:

- $u \leftarrow r_1 + h \cdot r_2$.
- $v \leftarrow \text{truncate}(s \cdot r_2 + e, \ell) + \mathcal{C}.\mathsf{Encode}(m)$.

With conventional multiplication, this would require two full polynomial multiplications. However, using the FFT-based method in the HQC-PKE.EncryptCache procedure from Algorithm 3, this process is significantly streamlined:

1. **Single Transform**: We compute $\hat{r_2}$ with only one input transform of r_2.
2. **Reuse**: This $\hat{r_2}$ is then reused, undergoing pointwise multiplication ($\otimes$) with the cached elements $\hat{h}$ and $\hat{s}$ in $\mathsf{ek}_{\mathsf{CPKE}}$, respectively.
3. **Inverse Transform**: Finally, the results of the pointwise multiplications are converted back to polynomial form using the inverse transform ($\texttt{addFFT}^{-1}$).

By sharing the transformed result of r_2, we save one input transform. When combined with the key caching in Subsection 4.1, the number of `addFFT` computations is minimized during encryption, and the dominant operations become two output transforms.

Due to the FO transform, the decapsulation operation includes a re-encryption step, meaning the aforementioned optimizations for encryption also effectively accelerate the overall performance of decapsulation.

4.3 Faster Implementation for RM-RS Codes

4.3.1 SIMD RM Decoding

The encoding of the RM code is straightforward, as it involves a simple matrix vector product using `XOR` and duplication of the message bits.

For decoding, while various algorithms exist—such as the Sidelnikov-Pershakov algorithm or recursive list decoding—we adopt the approach based on the Fast Hadamard Transform (FHT). This choice is made to comply with the official HQC specification [AAB+25], ensuring compatibility. Furthermore, the FHT's regular butterfly structure facilitates a constant-time implementation to mitigate timing side-channels and maps efficiently to the SIMD vector instructions focused on in this work.

The decoding algorithm in general goes through the following steps:

- `expand_and_sum`: expands codeword bits into computing units and sums the 3 or 5 multiple codeword copies into one.
- `Hadamard`: performs the Hadamard transform, the maximum-likelihood decoding algorithm on first-order RM codes, on the collected codewords.
- `find_peaks`: finds the location of the highest value in the Hadamard transform result.

We implemented a vectorized version of the sequential `expand_and_sum` procedure defined in [AAB+25]. Our SIMD approach efficiently processes data by broadcasting 2-byte data across all SIMD lanes and performing the required bit extraction using bit-masking. This vectorized `expand_and_sum` not only improves performance but also mitigates the power side-channel vulnerabilities demonstrated in [PRJB23, MNMD25], which specifically target the sequential bit splitting process.

4.3.2 Improving Multiplication in $\mathbb{F}_{256}$

The encoding and decoding of the RS code involves heavy multiplications in $\mathbb{F}_{256}$. The implementation in [AAB+25] uses `PCLMULQDQ` for the multiplication phase on the field in $\mathbb{F}_{256}$ to keep the time-constancy. To achieve higher performance, we instead implemented the SIMD $\mathbb{F}_{256}$ multiplication from [BCH+23] in our implementation. All multiplications in $\mathbb{F}_{256}$ are accomplished using table look-up instructions. Although the `GF2P8MULB` instruction in GFNI is designed for $\mathbb{F}_{256}$ arithmetic in the AES representation, we note that the affine transform instruction also available in GFNI provides an efficient method to change the field representation. This allows for the effective integration of `GF2P8MULB` into any $\mathbb{F}_{256}$ multiplication routine.

5 Implementation and Benchmarks

This section details our optimized implementations on the target platforms and presents their performance benchmarks. Subsection 5.1 reports the results of polynomial multiplication with `addFFT`. Subsection 5.2 reports our SIMD optimization for $\mathcal{C}$.Encode and $\mathcal{C}$.Decode. Subsection 5.3 reports the overall performance gain from our implementation.

Benchmark Platforms We evaluated the performance of our implementation across three distinct hardware architectures: a high-end x86 server (AMD EPYC), an Apple M1 system, and a low-power ARM embedded platform (Cortex-A72). The specifics of each platform and the build environment are detailed below.

- AMD EPYC 9754 (**X86Avx2** and **GFNI**): Benchmarks on the x86 platform were performed on an AMD EPYC 9754. This CPU supports the AVX2, `PCLMULQDQ`, and GFNI instruction sets. To evaluate the impact of instruction sets, we define two specific execution environments: AVX2+`PCLMULQDQ` (**X86Avx2**) and AVX2+`PCLMULQDQ` +GFNI (**GFNI**). All x86 benchmarks were performed on Ubuntu 24.04.2 LTS using GCC 13.3.0. Turbo Boost and Hyper-Threading (SMT) were disabled.

- Apple M1 (**AplM1**): We used an Apple M1 CPU running macOS 26.0.1. The code was compiled with Apple Clang 17.0.0.
- ARM Cortex-A72 (**Pi4**). This platform utilized a Raspberry Pi 4 featuring an ARM Cortex-A72 CPU. The environment was Debian GNU/Linux 12 with GCC 12.2.0.

All measurements report the median of 1000 measurements from random instances.

5.1 Polynomial Multiplication

5.1.1 Implementation

We implemented platform-specific optimization strategies to leverage the instruction set architectures.

X86Avx2 For polynomial multiplication in $\mathbb{F}_{2^{64}}$, we utilize the `PCLMULQDQ` instruction. The process involves two primary phases: 1) Multiplication: A single `PCLMULQDQ` performs the 64×64-bit polynomial multiplication, resulting in a 127-degree polynomial. 2) Modular Reduction: The result is then reduced modulo the field polynomial. This involves multiplying the terms with degrees greater than 63 by the module $x^4 + x^3 + x + 1$ with one `PCLMULQDQ`. As the intermediate result may still reach degree 67, a subsequent `PSHUFB` instruction is employed to reduce the terms of degree 64 to 67 down to degrees 0 to 7.

On optimizing the `addFFT` computation, we process two butterfly stages simultaneously, letting us perform fewer modular reduction operations over accumulating two products. Furthermore, we maximize instruction throughput and effectively hide the latency associated with field multiplication by processing four field multiplications in parallel.

GFNI We adopt the field of $\mathbb{F}_{256^{2^2}}$ as (4) in FAFFT, enabling us to leverage the `GF2P8MULB` instruction for native multiplication in $\mathbb{F}_{256}$. For `addFFT` optimization in this setting, the constant multipliers are consistently smaller than $\underline{2^{24}}$. This ensures the highest byte in the $\mathbb{F}_{256^{2^2}}$ field is always zero, allowing for faster field arithmetic by skipping the processing of the zero-valued byte.

AplM1 Our implementation and optimization strategy on the Apple M1 platform is similar to **X86Avx2** platform, as the ARM `PMULL.P64` instruction is functionally equivalent to the `PCLMULQDQ` instruction. The transition from 256-bit AVX registers to 128-bit NEON registers provided an additional benefit by reducing some data rearrangement overhead, which proved advantageous for the FAFFT implementation.

Pi4 On platforms lacking a dedicated long polynomial multiplication instruction, e.g. `PMULL.P64`, we adopted a multiplication algorithm based on short multiplication instructions. We implemented the KS+CRT method for polynomial multiplication, running the `addFFT` over the relatively small field $\mathbb{F}_{256^2}$ (as defined in Eq. (3)), which contrasts with the larger $\mathbb{F}_{2^{64}}$ or $\mathbb{F}_{256^{2^2}}$ fields used in FAFFT.

The `PMULL.P8` instruction performs 8×8-bit polynomial multiplication, making it suitable for the multiplication phase of field arithmetic in $\mathbb{F}_{256}$. The subsequent reduction phase is efficiently accomplished using two table look-up operations.

Furthermore, during the `addFFT` computation within the KS algorithm, the multiplier constants used in the butterfly operations are consistently in the $\mathbb{F}_{256}$ subfield in starting butterfly stages. This allows the multiplications in butterfly to become two $\mathbb{F}_{256}$ multiplications instead of a normal multiplication in $\mathbb{F}_{256^2}$, due to the carefully chosen field representation specified in Eq. (3).

On the CRT method, we need to compute the multiplication in the modular rings $\mathbb{F}_{2^{16}}[x]/x^{384}$ or $\mathbb{F}_{2^{16}}[x]/x^{1024}$. The multiplication in these rings is executed using a Karat-

Table 4: Cycle counts of polynomial multiplication

Platform	Algorithm	HQC-1 n =17 669	HQC-3 n =35 851	HQC-5 n =57 637
X86Avx2	Toom-Karatsuba ([AAB+25])	27 662	80 881	195 234
	FAFFT ($\mathbb{F}_{2^{64}}$)	57 612	139 629	202 792
	FAFFT ($\mathbb{F}_{256^{2^2}}$,**GFNI**)	34 042	87 652	102 997
AplM1	Toom-Karatsuba (porting [AAB+25])	31 359	93 172	236 753
	FAFFT ($\mathbb{F}_{2^{64}}$)	39 032	92 186	135 700
Pi4	Toom-Karatsuba (porting [AAB+25])	301 632	912 250	2 293 670
	KS + CRT ($\mathbb{F}_{256^2}$)	329 319	788 052	1 317 323

Table 5: Cycle counts of codec in HQC

Platform	Implementation	HQC-1		HQC-3		HQC-5	
		Encode	Decode	Encode	Decode	Encode	Decode
X86Avx2	[AAB+25]	2932	90 465	4037	139 102	8346	262 629
	This work	1271	64 062	1713	79 523	2650	158 793
	This work,**GFNI**	1219	63 157	1481	78 540	2519	157 719
AplM1	This work	1364	53 619	1797	69 152	2688	141 366
Pi4	This work	1243	134 793	1599	173 903	2500	360 966

suba algorithm layered over a base 8×8 schoolbook multiplication over $\mathbb{F}_{2^{16}}$. Since the input data is handled in byte chunks by the KS method, the product of two byte-chunk inputs does not trigger the reduction phase of $\mathbb{F}_{2^{16}}$ multiplication. Consequently, the `PMULL.P8` instruction is ideal for the base multiplication.

The reverse CRT map, which involves repeated multiplication and reduction steps, typically leads to frequent memory accesses to the two intermediate products. To address this overhead, we employ a product scan technique to minimize memory load and store operations. Instead of repeatedly accessing the full memory blocks, we sequentially collect all the intermediate coefficients contributing to the final result. Most product coefficients require only data rearrangement. For the remaining complex coefficients, we load only the essential data on demand, thus effectively avoiding redundant data transfers.

5.1.2 Benchmark Results

We report the performance of our FFT-based polynomial multiplication implementation in *Table* 4. The results indicate that the FFT-based approach outperforms Toom-Karatsuba for polynomials of sufficiently high degree, confirming the expected asymptotic advantage of the FFT method. For comparison on the **Pi4** platform, we adapted the Toom-Karatsuba polynomial multiplication algorithms provided by the official HQC team. Specifically, the $64 \times 64 \to 128$ base multiplication in $\mathbb{F}_2[x]$ was built using the schoolbook method detailed in [CGLD13], with Karatsuba then applied for higher-degree polynomials.

Comparative performance analysis Our evaluation focuses on three key comparisons to highlight the algorithmic advantages and the impact of instruction set architectures.

We begin by comparing the polynomial multiplication algorithms across the **X86Avx2** and **AplM1** platforms, which utilize functionally equivalent instructions for long polynomial multiplication (`PCLMULQDQ` and `PMULL.P64`). On **AplM1**, the benchmark results indicate a crossover point where FAFFT begins to outperform Toom-Karatsuba, occurring between

HQC-1 and HQC-3. Conversely, on **X86Avx2**, Toom-Karatsuba retains a slight advantage, outperforming FAFFT by 6% even at the largest scale, HQC-5. This result suggests that the two algorithms achieve a near-even break in the context of HQC's required polynomial degrees, given the availability of dedicated long polymul instructions. The relative performance advantage appears to be highly sensitive to the underlying microarchitecture.

Next, we quantify the performance effect of the GFNI instruction set by comparing the results between the **X86Avx2** and **GFNI** platforms. While both implementations utilize the same FAFFT algorithm, they operate over two distinct fields: $\mathbb{F}_{2^{64}}$ and $\mathbb{F}_{256^{2^2}}$, respectively. The results show that the specialized GFNI instruction set provides a significant performance improvement over pure `PCLMULQDQ`, yielding a performance gain of approximately 100% for HQC-5. We emphasize that this improvement is not a naive benefit of a powerful new ISA. Instead, the gain stems from the co-design of the algorithm and the hardware instructions. To fully utilize the `GF2P8MULB` instruction within GFNI, we specifically adopted the $\mathbb{F}_{256^{2^2}}$ field and its representation in Eq. (4) to minimize the cost of constant multiplication during the Butterfly process.

Finally, on the resource-constrained **Pi4** platform, which is equipped only with the `PMULL.P8` instruction for multiplication, we observe a distinct crossover point, where the KS+CRT algorithm begins to surpass Toom-Karatsuba, occurring approximately near the complexity of HQC-1. This result indicates that on resource-limited embedded systems, the FFT-based approach proves more efficient than the Toom-Karatsuba algorithms for the polynomial degrees used in HQC. This finding aligns strongly with prior research. The study by [LJKH25] demonstrated that a bit-sliced implementation of the FAFFT algorithm is more efficient than Karatsuba algorithms across all HQC parameters on embedded systems lacking dedicated polynomial multiplication instructions.

Addressing portability and performance attribution Our evaluation clarifies the performance gains attributable to portable algorithmic improvements versus platform-specific instruction sets. The performance comparison between **X86Avx2** and **AplM1**, which rely on functionally equivalent `PCLMULQDQ` instructions, demonstrates the portability of the FFT-based approaches. However, the 100% gain observed in **GFNI** is attributable to the non-portable GFNI (`GF2P8MULB`) instruction set. This highlights that while the core FFT algorithm is portable, maximum performance is achieved through dedicated algorithm-ISA co-design. The performance dominance observed on the resource-constrained **Pi4** platform further validates the portability and effectiveness of the FFT-based approach even in the absence of any long-multiplication instruction.

5.2 SIMD Code Implementation

Table 5 summarizes our SIMD-optimized RM-RS codec, confirming the effectiveness of SIMD across all tested platforms. Our proposed optimizations achieve substantial speedups over the official implementation [AAB+25] on **X86Avx2**, notably providing an approximate 1.66× acceleration for the HQC-5 decoding operation. These performance gains are attributed to two key SIMD strategies: optimized $\mathbb{F}_{256}$ multiplication and vectorized bit extraction in the `expand_and_sum` procedure. Furthermore, the vectorized bit extraction mitigates side-channel vulnerabilities. The $\mathbb{F}_{256}$ multiplication is further enhanced on **GFNI** platforms using AESMUL, providing an additional performance reduction. Our implementation also demonstrates strong scalability on `ARMv8-A`, with the **Pi4** platform validating the `PMULL.P8`-based method's effectiveness on resource-limited embedded systems.

Table 6: Kilo-cycle counts of HQC implementations

Platform	Implementation	HQC-1	HQC-3	HQC-5
		Key/Enc/Dec	Key/Enc/Dec	Key/Enc/Dec
X86Avx2	[AAB+25]	72 / 141 / 315	173 / 340 / 671	353 / 697 / 1344
	This work	103 / 187 / 363	231 / 427 / 761	358 / 648 / 1197
	cache key	117 / 139 / 292	263 / 330 / 610	423 / 463 / 910
	This work, **GFNI**	78 / 144 / 296	179 / 339 / 622	249 / 464 / 901
	cache key	87 / **110** / **245**	200 / **272** / **517**	292 / **343** / **711**
AplM1	Porting [AAB+25]	72 / 155 / 303	186 / 398 / 688	412 / 842 / 1445
	This work	80 / 157 / 313	188 / 374 / 672	308 / 588 / 1092
	cache key	93 / **125** / **259**	212 / **302** / **554**	352 / **460** / **870**
Pi4	Porting [AAB+25]	416 / 820 / 1391	1191 / 2342 / 3735	2798 / 5551 / 8725
	This work	447 / 784 / 1374	1078 / 1903 / 3137	1801 / 3185 / 5336
	cache key	547 / **568** / **1029**	1279 / **1465** / **2399**	2278 / **2348** / **3986**

5.3 Optimized Implementations of HQC

Table 6 compares our HQC implementations, which incorporate FFT-based polynomial multiplication and SIMD codec optimizations, against the official optimization [AAB+25]. The results demonstrate that the FFT-based approach, when combined with the cache key strategy, outperforms the HQC official optimization in both encapsulation and decapsulation across all tested platforms, despite exhibiting slower raw multiplication speeds in the HQC-1 implementation. The performance gain is distinctly two-fold: First, the GFNI-enabled implementation shows the most significant acceleration, leading a clear performance gap over x86 platforms (e.g., a 1.68× speedup on HQC-5 **Dec** over our base **X86Avx2** implementation). This emphasizes the impact of algorithm-ISA co-design. Second, the cache key strategy shifts the input transform cost from the frequent **Enc** and **Dec** operations to the one-time **KeyGen** operation. This strategy results in the best cycle counts for **Enc** and **Dec** operations across all security levels, regardless of the underlying microarchitecture.

Performance profiling of HQC KEM We conduct a more thorough cost comparison for the four major computational components in the HQC KEM. Table 7 shows the detailed cycle counts consumed by these major components. The results from the reference implementation of the HQC team indicate that ring multiplication is indeed the computational bottleneck in HQC. This situation is particularly severe on platforms lacking dedicated hardware instructions, where only the reference code can be executed. Conversely, the profiling data from our optimized implementation demonstrates that the optimization of ring operations significantly reduces their percentage of the total execution time. However, even after optimizing the ring operation with dedicated instructions, it still accounts for the largest fraction of the total running time, affirming that it remains the most critical target for future optimization efforts.

6 Concluding Remarks

In this work, we demonstrated that FFT-based multiplication can significantly accelerate the HQC key encapsulation mechanism. Our concluding findings are twofold:

First, the primary performance advantage of our approach does not necessarily come from the raw speed of FFT-based multiplication, which are actually comparable to highly-optimized Toom-Karatsuba on platforms on HQC's settings of polynomial degrees. Instead,

Table 7: Kilo-cycle counts for major operations in HQC

		Official Reference[AAB+25]				The GFNI Implementation			
HQC	**Operation**	**Hash**	**Sample**	**Ring**	**Code**	**Hash**	**Sample**	**Ring**	**Code**
HQC-1	Keygen	1.8	104.6	3612.8	0.0	1.5	40.5	33.4	0.0
	Encap	52.3	142.6	7222.4	27.0	44.1	37.8	59.6	1.2
	Decap	146.0	142.1	10829.6	347.1	97.8	37.4	92.8	64.3
HQC-3	Keygen	1.9	283.3	10988.2	0.0	1.5	87.4	87.6	0.0
	Encap	102.7	407.6	21954.2	42.9	86.8	86.0	161.7	1.5
	Decap	328.8	406.9	32933.2	473.2	199.2	85.5	249.6	80.0
HQC-5	Keygen	2.2	552.5	27081.2	0.0	1.5	150.1	95.1	0.0
	Encap	162.6	828.9	54100.9	101.8	136.9	157.4	161.1	2.5
	Decap	574.1	827.9	81132.9	1059.0	321.6	156.9	255.9	160

the most substantial gains are achieved by eliminating redundant computations. By caching the input transforms of key pairs and reusing the transform of the common operand r_2 during encryption, we effectively reduce the computational overhead in both Encap and Decap. As evidenced by our benchmarks, this strategy leads to better performance, particularly for HQC-1, even when the multiplication itself is not the fastest.

Second, our results show that the optimal choice of a polynomial multiplication algorithm is highly dependent on the target architecture's available instructions, more so than on asymptotic complexity alone. We conclude with the following recommendations:

- For platforms with long multiplication instruction (e.g., `PCLMULQDQ` and `PMULL.P64`): The performance between FAFFT and Toom-Karatsuba is characterized by a near-even break for HQC-sized polynomials. Specifically, the FAFFT shows an advantage on the Apple M1 microarchitecture (with the crossover point near HQC-1/HQC-3), while Toom-Karatsuba maintains a slight edge on the AMD EPYC (x86) microarchitecture at larger scales. This indicates that the relative performance superiority is highly sensitive to the underlying microarchitectural implementation. However, our work shows that even when raw multiplication is slower, an FFT-based implementation yields superior overall KEM performance in encapsulation and decapsulation due to significant savings from transform caching and reuse.

- For platforms with only short multiplication instructions (e.g., `PMULL.P8`): On these architectures, the crossover point occurs at a lower polynomial degree. Our findings clearly show that the additive FFT approach, combined with CRT, becomes the superior strategy for the HQC-3 and HQC-5 parameter sets. While Toom-Karatsuba holds a narrow lead in raw multiplication for the HQC-1, the FFT-based approach still delivers better overall KEM performance when caching is applied, making it the more effective strategy.

Acknowledgements

We are 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

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.

[ABP25] Francesco Antognazza, Alessandro Barenghi, and Gerardo Pelosi. An Efficient and Unified RTL Accelerator Design for HQC-128, HQC-192, and HQC-256 . *IEEE Transactions on Computers*, 74(07):2306–2320, July 2025.

[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.

[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.

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

[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.

[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.

[CGLD13] Danilo Câmara, Conrado P. L. Gouvêa, Julio López, and Ricardo Dahab. Fast software polynomial multiplication on arm processors using the neon engine. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, pages 137–154, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[CYW25] Chun-Ming Chiu, Bo-Yin Yang, and Bow-Yaw Wang. A new trick for polynomial multiplication: A verified crt polymul utilizing a monomial factor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):795–816, Sep. 2025.

[DXN+24] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. Fast and efficient hardware implementation of HQC. In Claude Carlet, Kalikinkar Mandal, and Vincent Rijmen, editors, *SAC 2023*, volume 14201 of *LNCS*, pages 297–321, Fredericton, Canada, August 14-18, 2024. Springer, Cham, Switzerland.

[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.

[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.

[Int14] Intel. Intel carry-less multiplication instruction and its usage for computing the gcm mode, 2014. https://cdrdv2-public.intel.com/836172/clmul-wp-rev-2-02-2014-04-20.pdf.

[Int21] Intel. Galois field new instructions (gfni) technology guide, 2021. https://cdrdv2.intel.com/v1/dl/getContent/644497?fileName=GFNI_TechGuide_644497v2.pdf.

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

[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.

[MNMD25] Nathan Maillet, Cyrius Nugier, Vincent Migliore, and Jean-Christophe Deneuville. Key recovery from side-channel power analysis attacks on non-SIMD HQC decryption. In Yael Tauman Kalai and Seny F. Kamara, editors,

CRYPTO 2025, Part V, volume 16004 of *LNCS*, pages 70–102, Santa Barbara, CA, USA, August 17–21, 2025. Springer, Cham, Switzerland.

[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.

[PRJB23] Thales Paiva, Prasanna Ravi, Dirmanto Jap, and Shivam Bhasin. Et tu, brute? SCA assisted CCA using valid ciphertexts - A case study on HQC KEM. Cryptology ePrint Archive, Report 2023/1626, 2023.

[RLC+25a] Antonio Ras, Antoine Loiseau, Mikael Carmona, Simon Pontié, Guénaël Renault, Benjamin Smith, and Emanuele Valea. Optimizing hqc using frobenius additive fft on a risc-v-based system-on-chip. NIST Sixth PQC Standarization Conference, 2025. available at https://csrc.nist.gov/csrc/media/events/2025/sixth-pqc-standardization-conference/optimizing%20hqc%20using%20frobenius%20additive%20fft.pdf.

[RLC+25b] 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.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[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.